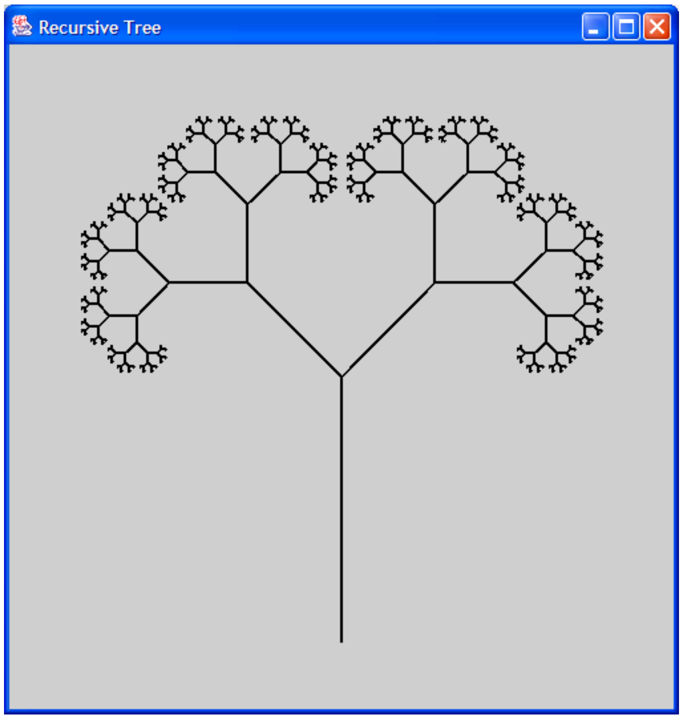
Introduction To Computer Science
An Object-Oriented Approach
Using Java 5

BlueJ and BeanShell Edition

Barry G. Adams
Department of Mathematics and Computer Science
Laurentian University



Contents

1 Introduction to Computation **1**
1.1 Introduction . . . 2
1.2 Algorithms . . . 2
1.3 Processors . . . 3
1.3.1 Functional units of a CPU . . . 4
1.4 Memory and I/O devices . . . 4
1.5 Programs . . . 6
1.6 Computer languages . . . 6
1.6.1 Machine and assembly languages . . . 7
1.6.2 High-level languages . . . 8
1.7 Translation and interpretation of programs . . . 9
1.8 Java virtual machine . . . 10
1.9 Java source code to bytecode translation example . . . 11
1.10 Review exercises . . . 12

2 Fundamental Data Types **13**
2.1 Fundamental data types and variables . . . 14
2.1.1 Integer and floating point data types . . . 14
From mathematical to computer data types . . . 15
The char data type . . . 16
The boolean data type . . . 16
2.1.2 Integer and floating point literals . . . 16
2.1.3 Declaring and initializing variables in Java . . . 17
Rules for naming variables . . . 19
Constants . . . 19
2.2 Arithmetic operations and expressions . . . 20
2.2.1 Basic arithmetic operations . . . 20
2.2.2 Arithmetic expressions and precedence rules . . . 20
2.3 Assignment statements . . . 21
2.3.1 Try it with BeanShell . . . 22
2.4 Conversion between numeric types (type casting) . . . 24
2.4.1 Truncation of floating point numbers . . . 25
2.4.2 Loss of precision in floating point conversions . . . 26
2.5 Arithmetic functions from the `Math` class . . . 27

2.5.1 Examples of athematical functions . . . 27
2.5.2 Rounding floating point numbers . . . 29
2.5.3 Mathematical function prototypes . . . 30
2.6 Terminology introduced in this chapter . . . 31
2.7 Review exercises . . . 34
2.8 BeanShell exercises . . . 36

3 Writing Simple Classes **39**
3.1 Introduction . . . 40
3.2 `CircleCalculator` class using BlueJ . . . 40
3.2.1 Experimenting with the class . . . 40
3.2.2 `CircleCalculator` source code . . . 43
3.2.3 Explanation of the source code . . . 43
Class declaration . . . 44
Instance data fields . . . 44
Constructor declaration . . . 44
Method declarations . . . 45
3.3 `TriangleCalculator` class using BlueJ . . . 46
3.3.1 Experimenting with the class . . . 46
3.3.2 `TriangleCalculator` source code . . . 48
3.3.3 Explanation of the source code . . . 50
Class declaration . . . 50
Instance data fields . . . 50
Constructor declaration . . . 50
Method declarations . . . 51
3.3.4 Testing `TriangleCalculator` . . . 51
3.4 `QuadraticRootFinder` class using BlueJ . . . 52
3.4.1 Experimenting with the class . . . 52
3.4.2 `QuadraticRootFinder` source code . . . 54
3.4.3 Explanation of the source code . . . 55
Class declaration . . . 55
Instance data fields . . . 55
Constructor declaration . . . 56
Method declarations . . . 56
3.4.4 Testing `QuadraticRootFinder` . . . 57
3.5 Using BeanShell with objects . . . 58
3.5.1 Constructor call expressions . . . 58
3.5.2 Method call expressions . . . 59
3.5.3 BeanShell examples . . . 60
3.6 Writing and viewing Javadoc class documentation . . . 62
3.6.1 Javadoc rules . . . 63
3.6.2 Javadoc version of `CircleCalculator` . . . 63
Class `CircleCalculator` . . . 63
3.6.3 Javadoc version of `TriangleCalculator` . . . 64

Class `TriangleCalculator` . . . 64
3.6.4 Javadoc version of `QuadraticRootFinder` . . . 67
Class `QuadraticRootFinder` . . . 67
3.6.5 Viewing the documentation . . . 69
3.6.6 Implementation and documentation views . . . 70
3.6.7 Project documentation . . . 70
3.7 Syntax and logical errors . . . 71
3.7.1 Some common syntax errors . . . 73
Forgetting a semi-colon . . . 73
Undeclared variables . . . 73
Declaring a variable more than once . . . 74
Misspelling the constructor name . . . 75
Forgetting `new` in constructor call expressions . . . 75
3.7.2 Some common logical errors . . . 75
Using an incorrect formula . . . 76
Redeclaring an instance variable . . . 76
Using a return type on a constructor . . . 76
3.7.3 Invoking a method on a non-existent object . . . 77
3.8 Summary of terminology . . . 78
3.9 Review exercises . . . 86
3.10 Programming exercises . . . 86

4 Classes, Objects, and Methods 91
4.1 Introduction . . . 92
4.2 `String` class . . . 92
4.2.1 Constructing strings . . . 92
4.2.2 String expressions and concatenation . . . 93
String expressions containing numbers . . . 94
4.2.3 String methods . . . 95
The length of a string . . . 95
Converting a number to a string . . . 96
Extracting a single character from a string . . . 96
Constructing a substring . . . 96
Trimming a string . . . 97
Upper case and lower case conversions . . . 97
Searching for substrings . . . 98
4.2.4 Displaying numbers and strings . . . 99
Using `System.out` in BeanShell . . . 99
Special characters in strings . . . 99
Using `System.out` in BlueJ . . . 100
The `toString` method . . . 101
Defining our own `toString` method . . . 101
4.2.5 Formatting numbers and strings (Java 5) . . . 102
4.3 Example classes that use the `String` class . . . 104

4.3.1 `BankAccount` class (first version) . . . 104
Designing the class . . . 104
Implementing the class . . . 105
Class `BankAccount` . . . 106
Testing the class . . . 108
4.3.2 `InitialsMaker` class . . . 109
Designing the class . . . 109
Implementing the class . . . 110
Class `InitialsMaker` . . . 110
Testing the class . . . 111
4.3.3 `PasswordGenerator` class . . . 111
Designing the class . . . 111
Implementing the class . . . 112
Class `PasswordGenerator` . . . 113
Testing the class . . . 115
The “`this`” object . . . 116
4.4 Association and aggregation . . . 117
4.4.1 Association . . . 117
4.4.2 Aggregation . . . 117
4.4.3 `TriangleCalculatorTester` class . . . 117
Class `TriangleCalculatorTester` . . . 118
4.4.4 `Point` class . . . 118
Designing the class . . . 119
Implementing the class . . . 119
Class `Point` . . . 119
Testing the class . . . 121
4.4.5 `Circle` class . . . 121
Designing the class . . . 121
Implementing the class . . . 122
Class `Circle` . . . 122
Testing the class with BeanShell . . . 124
Testing the class with BlueJ . . . 125
Class `CircleTester` . . . 127
4.5 Other library classes . . . 127
4.5.1 Dates and times . . . 127
`Date` class . . . 127
`SimpleDateFormat` class . . . 128
`Calendar` class . . . 129
`Person` class that uses `Calendar` . . . 131
Class `Person` . . . 131
Specialized `Calendar` class . . . 132
Class `CalendarMonth` . . . 133
Explanation of the class . . . 136
Testing the class . . . 136

Class `CalendarMonthTester` 137
4.5.2 Currency formatting 138
4.5.3 Formatting fixed and floating point numbers (Java 1.4) 139
4.6 Review of OOP concepts 140
4.6.1 Constructing objects 140
Using a constructor 140
Using a static factory method 140
Using `this` as a constructor call expression 141
Default constructor 142
4.6.2 Object references 143
Null references 144
Comparison of primitive and reference types 145
Why do we need both primitive and reference types? 146
Assignment statements for reference types 146
4.6.3 Using references as arguments and method return values 147
4.6.4 Data encapsulation and integrity 148
Side-effects 149
Class `MPoint` 149
Class `MCircle` 151
Copy constructor 155
Class `MCircle` 156
`BankAccount` example 157
Class `TransferAgent` 158
4.6.5 Instance variables and methods 160
Instance variables 160
Instance methods 160
Using an instance method 160
Method composition 162
4.6.6 Static variables, constants, and methods 162
Static variables 162
Static constants 162
Static methods 162
Using static methods 163
Counting the number of objects created from a class 163
4.6.7 Kinds of variables and arguments 165
4.6.8 Call by value argument passing mechanism 165
Call by value for an argument of primitive type 166
Class `ArgumentTester1` 166
Call by value for an argument of reference type (object type) 167
Class `ArgumentTester2` 168
4.6.9 `main` method 169
Adding a `main` method to a class 169
Class `CircleTester` 170
Writing a runner class 171

Class `BankAccountRunner` . . . 171
4.7 Running a class with a `main` method . . . 172
4.8 Review exercises . . . 172
4.9 BeanShell exercises . . . 173
4.10 Programming exercises . . . 175

5 Using Graphics Classes and Objects **181**
5.1 Introduction . . . 182
5.2 Using the `GraphicsFrame` class . . . 182
5.2.1 `EmptyDrawing` template for simple graphics programs . . . 183
Class `EmptyDrawing` . . . 183
5.3 The graphics context . . . 185
5.3.1 `Graphics` and `Graphics2D` objects . . . 185
5.3.2 `paintComponent` method . . . 185
5.4 User space and device space . . . 186
5.5 Graphics classes and objects . . . 187
5.5.1 `Point2D` and `Line2D` classes . . . 187
Defining points . . . 188
Defining lines using points . . . 188
Class `DrawLine` . . . 189
Drawing rectangles using lines . . . 190
Class `DrawBiggestRectangle` . . . 190
Defining lines using coordinates . . . 191
Resizing the drawing window . . . 192
Class `DrawBiggestRectangle2` . . . 193
5.5.2 Geometrical Shape Hierarchy (`java.awt.geom` package) . . . 194
Drawing points . . . 194
5.5.3 `RectangularShape` classes . . . 195
`Rectangle2D` class . . . 196
Class `DrawBiggestRectangle3` . . . 196
`Ellipse2D` class . . . 197
Class `DrawEllipse` . . . 197
`RoundRectangle2D` class . . . 198
Class `DrawRoundRectangle` . . . 198
`Arc2D` class . . . 199
Class `DrawArc` . . . 200
5.6 The drawing process . . . 202
5.6.1 Specifying attributes . . . 202
Improving the rendering quality (smoothness) . . . 202
Specifying colors . . . 203
Specifying line thickness . . . 205
Class `DrawArc2` . . . 205
Filling and stroking . . . 206
5.7 Put on a happy face . . . 207

5.7.1 Designing the face with boxes . . . 207
5.7.2 No-frills happy face . . . 207
Class `FaceMaker1` . . . 208
5.7.3 Colorful happy face . . . 210
Class `FaceMaker2` . . . 211
5.7.4 Facial transformations . . . 213
Affine transformations . . . 213
Resizing the face to fit the window . . . 213
Class `FaceMaker3` . . . 214
Making a half-size face . . . 217
Rotated happy face . . . 217
5.7.5 Four happy faces for the price of one . . . 218
Top-level description . . . 218
Drawing one face . . . 220
Defining the face geometry . . . 221
Transforming a face . . . 221
Rendering a face . . . 222
Class `FaceMaker6` . . . 222
5.7.6 Running the six face maker programs together . . . 225
Class `AllTogether` . . . 225
5.8 Making your own coordinate transformations . . . 225
5.8.1 Specific transformations . . . 226
Class `BarGraph1` . . . 228
5.8.2 World to default user transformation . . . 229
5.8.3 Coordinate system class . . . 231
Class `CoordinateSystem` . . . 232
5.8.4 Drawing a bar graph . . . 233
Class `BarGraph2` . . . 233
5.8.5 Drawing a regular pentagon . . . 235
Class `DrawPentagon1` . . . 235
5.8.6 General transformation using affine transformations . . . 237
Bar graph using an affine transformation . . . 238
Class `BarGraph3` . . . 239
Pentagon using an affine transformation . . . 241
Class `DrawPentagon2` . . . 241
5.8.7 Transforming individual shapes . . . 243
Class `DrawPentagon3` . . . 244
5.9 Review exercises . . . 246
5.10 Programming exercises . . . 246

6 Making Decisions **251**
6.1 Introduction . . . 252
6.2 Simple boolean expressions . . . 252
6.3 If-statements . . . 254

6.4 Real roots of a quadratic equation . . . 256
6.4.1 `QuadraticRootFinder` class . . . 256
Class `QuadraticRootFinder` . . . 256
6.5 Block declaration of variables . . . 257
6.6 If-statement with no else . . . 258
6.7 Comparison of floating point numbers . . . 260
6.7.1 Floating point tester class . . . 260
Class `FloatingPointTester1` . . . 260
6.8 Conditional operator . . . 262
6.9 Nested and multiple (N-way) if-statements . . . 263
6.10 Common errors with if-statements . . . 265
6.11 Compound boolean expressions . . . 268
6.11.1 Writing expressions using AND, OR, and NOT . . . 268
Operator precedence rules . . . 269
DeMorgan's laws . . . 270
Testing numerical ranges . . . 270
6.11.2 Leap year problem . . . 271
6.11.3 Short circuit evaluation . . . 273
6.12 String comparison and equality . . . 273
6.12.1 Equals method for string omparison . . . 274
6.12.2 Lexicographical ordering of strings . . . 275
Class `CharacterDecoder` . . . 275
Using character codes to order strings . . . 276
6.12.3 `compareTo` method for string comparison . . . 276
Class `StringComparer` . . . 277
6.12.4 Case insensitive string comparison . . . 278
6.13 Boolean valued methods . . . 278
6.14 Error checking techniques . . . 279
6.14.1 Reporting errors . . . 280
6.14.2 Using boolean return values and exit to report errors . . . 281
6.15 Error reporting using exceptions . . . 282
6.15.1 Exception classes and objects . . . 282
6.15.2 Throwing exceptions in the `BankAccount` class . . . 282
Class `BankAccount` . . . 284
Class `ExceptionTester` . . . 286
6.15.3 Catching exceptions . . . 286
Class `ExceptionCatcher` . . . 287
6.16 Paper, scissors, rock game (PSR) . . . 288
6.16.1 Rules of the game . . . 288
6.16.2 Object-oriented PSR game . . . 288
Designing the `PSRPlayer` class . . . 289
Designing the `PSRGame` class . . . 289
`PSRPlayer` implementation . . . 289
Class `PSRPlayer` . . . 290

PSRGame implementation . . . 290
Class PSRGame . . . 291
Testing the class with BlueJ . . . 293
Testing the class with BeanShell . . . 293
Running the game using a main method . . . 293
6.17 Console Input Using a Scanner object . . . 294
6.17.1 Some useful Scanner methods . . . 294
6.17.2 One input per line input model . . . 295
6.17.3 Console interface class for the PSR game . . . 295
Class PSRGameRunner . . . 295
6.18 Complex roots of a quadratic equation . . . 297
6.18.1 Complex class . . . 297
Class Complex . . . 297
Class ComplexQuadraticRootFinder . . . 298
Testing the class with BlueJ . . . 301
Testing the class with BeanShell . . . 301
Console interface . . . 302
Class ComplexRunner . . . 302
6.19 Review exercises . . . 303
6.20 BeanShell exercises . . . 304
6.21 Programming exercises . . . 305

7 Repetition Structures 313
7.1 Introduction . . . 314
7.2 The while-statement (while-loop) . . . 314
7.2.1 Converting a digit string to an integer . . . 317
Class StringToIntConverter . . . 318
Class StringToIntRunner . . . 319
7.2.2 Square root algorithm using a while-loop . . . 319
Class SquareRootCalculator . . . 320
Class SquareRootRunner . . . 321
7.2.3 Double your money problem . . . 322
Class DoubleYourMoney . . . 323
Class DoubleYourMoneyRunner . . . 323
7.2.4 Factorization of an integer . . . 324
Class Factorizer . . . 325
Class FactorizerRunner . . . 326
7.3 Sentinel-controlled while-loops . . . 327
7.3.1 AverageMarkCalculator class . . . 328
Class AverageMarkCalculator . . . 328
7.4 Query-controlled while-loops . . . 328
7.4.1 BankAccount example . . . 329
Class MaxBalanceCalculator . . . 330
7.5 Do-while statement (do-while loop) . . . 331

7.6 General loop structures . . . 333
7.7 For-statement (for-loop) . . . 334
7.7.1 Pseudo-code for-loops for counting in steps . . . 334
7.7.2 For-loops for counting in steps . . . 335
7.8 Computing factorials . . . 337
7.8.1 Computing the factorial of an integer . . . 337
Class `FactorialCalculator` . . . 338
Class `FactorialRunner` . . . 338
7.8.2 Computing factorials using the `BigInteger` class . . . 339
Class `BigFactorialCalculator` . . . 341
Class `BigFactorialRunner` . . . 341
7.9 Expressing the for-loop as a while-loop . . . 342
7.10 Loan repayment table . . . 343
7.10.1 Right justifying numbers in a field of given width . . . 344
7.10.2 `StringBuilder` class . . . 345
7.10.3 Loan repayment table class . . . 345
Class `LoanRepaymentTable` . . . 346
7.10.4 Console user interface . . . 348
Class `LoanRepaymentTableRunner` . . . 348
7.11 Nested loops . . . 349
7.11.1 Investment table . . . 352
Class `InvestmentTable` . . . 353
7.11.2 Console user interface . . . 354
Class `InvestmentTableRunner` . . . 354
7.12 Plotting the graph of a function . . . 356
7.12.1 `SineGraph` class . . . 358
Class `SineGraph` . . . 358
7.13 Recursion and loops . . . 361
7.13.1 What is recursion? . . . 361
7.13.2 Examples of recursive definitions . . . 361
7.13.3 Recursive factorial method . . . 362
Class `FactorialCalculator` . . . 363
Class `FactorialRunner` . . . 363
7.13.4 Recursive gcd method . . . 364
Class `GcdCalculator` . . . 364
Class `GcdRunner` . . . 365
7.13.5 Non-recursive and recursive sum methods . . . 366
7.14 Common loop errors . . . 366
7.14.1 Misplaced semi-colon . . . 367
7.14.2 Off by one errors . . . 367
7.15 BeanShell exercises . . . 367
7.16 Programming exercises . . . 368

8 Array Data Types 379

8.1 Introduction . . . 380
8.2 Mathematical sequences and subscript notation . . . 380
8.3 Declaring and constructing arrays . . . 381
8.3.1 Arrays of primitive type . . . 382
Declaring array types . . . 382
Constructing arrays . . . 382
Assigning values to array elements . . . 382
Using array initializers . . . 383
8.3.2 Calculating the number of days in a month . . . 384
Class `DaysInMonthCalculator` . . . 384
Class `DaysInMonthRunner` . . . 385
Declaring the size of an array at run-time . . . 386
The length of an array . . . 386
8.3.3 Sequential array processing . . . 386
8.3.4 Arrays of object type . . . 387
`Point2D` arrays . . . 389
8.3.5 String arrays . . . 390
Command-line arguments . . . 390
Class `CommandLineArguments` . . . 391
Class `LoanRepaymentTableRunner` . . . 391
Class `InvestmentTableRunner` . . . 392
8.3.6 Using arrays as method arguments and return values . . . 393
Testing an array processing method . . . 395
Class `Average` . . . 395
Class `AverageRunner` . . . 396
8.4 Some simple array algorithms . . . 397
8.4.1 Algorithm for the maximum array element . . . 397
Class `MaxFinder` . . . 397
Class `MaxFinderRunner` . . . 398
8.4.2 Linear search slgorithm . . . 400
Class `LinearSearcher` . . . 400
Class `LinearSearcherRunner` . . . 401
8.4.3 Bubble sort algorithm . . . 402
Sorting an array of numbers . . . 403
Class `BubbleSorter` . . . 404
Class `BubbleSorterRunner` . . . 405
Sorting an array of strings . . . 406
8.5 Efficient evaluation of a polynomial . . . 406
8.5.1 Horner’s algorithm . . . 407
8.5.2 A class for polynomials . . . 408
Class `Polynomial` . . . 408
8.5.3 Testing the `Polynomial` class . . . 409
Class `PolynomialRunner` . . . 410
8.6 Line graph example using arrays . . . 411

8.6.1 Line graph class . . . 411
Choosing a coordinate system . . . 412
Drawing the axes . . . 413
Drawing the line segments . . . 414
Drawing the circles at each vertex . . . 414
Class `LineGraph` . . . 414
Class `SimpleTester` . . . 417
8.6.2 Drawing a random line graph . . . 418
Class `RandomTester` . . . 419
8.6.3 Converting arrays to `GeneralPath` objects . . . 420
8.7 For-each loop . . . 420
8.8 Methods with a variable number of arguments . . . 422
8.9 Two-dimensional arrays . . . 423
8.9.1 Multiplying matrices . . . 425
Class `MatrixMultiplier` . . . 426
Testing the method in BlueJ . . . 427
Testing the method in BeanShell . . . 428
Testing matrix multiplication . . . 429
Class `MatrixMultiplierRunner` . . . 429
8.9.2 Board games . . . 431
8.10 Card shuffling and dealing application . . . 432
8.10.1 `Card` class . . . 432
`Card` class design . . . 433
`Card` class implementation . . . 433
Class `Card` . . . 433
8.10.2 `CardDeck` class . . . 436
`CardDeck` class design . . . 436
`CardDeck` implementation . . . 437
Class `CardDeck` . . . 438
Testing the class in BlueJ . . . 440
Testing the class in BeanShell . . . 441
Testing the class from the command line . . . 441
Class `CardDeckTester` . . . 441
8.11 Review exercises . . . 443
8.12 BeanShell exercises . . . 443
8.13 Programming exercises . . . 444

9 Inheritance and Interfaces 451
9.1 Introduction . . . 452
9.2 What is inheritance? . . . 453
9.2.1 The “is-a” and “has-a” relationships . . . 454
9.3 Rules for declaring subclasses . . . 455
9.4 Simple examples of subclasses . . . 456
9.4.1 Graphics programs . . . 456

9.4.2 Extending a circle calculator class . . . 457
Class `CircleCalculatorA` . . . 457
Class `CircleCalculatorB` . . . 458
Class `CircleCalculatorTester` . . . 459
`BlueJ` project for the circle calculator classes . . . 460
9.4.3 Extending the `BankAccount` class . . . 460
Class `BankAccount` . . . 461
Class `JointBankAccount` . . . 464
`BlueJ` project for the bank account classes . . . 465
9.5 Polymorphism . . . 466
9.5.1 Polymorphic types . . . 466
Polymorphic types in the `BankAccount` hierarchy . . . 466
Class `AccountTester` . . . 467
Examples of polymorphism . . . 468
9.5.2 Polymorphic methods . . . 469
A polymorphic bank account transfer method . . . 469
The polymorphic `toString` method . . . 470
Class `AccountTester2` . . . 470
Compile-time and run-time types . . . 471
9.6 Abstract classes and polymorphism . . . 471
9.6.1 An employee inheritance hierarchy . . . 471
9.6.2 `Employee` and `Manager` classes . . . 472
Class `Employee` . . . 472
Class `Manager` . . . 473
9.6.3 Polymorphism in the `Employee` hierarchy . . . 474
Class `EmployeeProcessor` . . . 474
9.7 The `Object` class . . . 476
9.7.1 Overriding `Object` class methods . . . 476
Overriding the `toString` method . . . 476
Overriding the `equals` method . . . 477
Class `PointEqualsTester` . . . 477
9.8 Final classes . . . 479
9.9 Interfaces . . . 479
9.9.1 Implementing the `Measurable` interface . . . 482
A measurable circle class . . . 482
Class `Circle` . . . 482
A measurable rectangle class . . . 484
Class `Rectangle` . . . 484
Polymorphism with the `Measurable` interface . . . 485
Class `MeasurableTester` . . . 485
9.9.2 Polymorphism with the `Shape` interface . . . 486
Class `ShapeTester` . . . 487
9.10 Multiple interfaces . . . 488
9.10.1 Interface specifications . . . 489

Interface `Measurable` . . . 489
Interface `Translatable` . . . 489
Interface `Scalable` . . . 490
9.10.2 Classes that implement the interfaces . . . 490
Class `Circle` . . . 490
Class `Rectangle` . . . 491
9.10.3 Typecasts with multiple interfaces . . . 492
Class `MultipleInterfaceTester` . . . 492
9.11 Implementing the `Shape` interface . . . 493
9.11.1 `Shape` interface methods . . . 493
9.11.2 Extending a `ShapeAdapter` class which implements `Shape` . . . 494
Class `ShapeAdapter` . . . 494
9.11.3 `Triangle2D` class that uses `ShapeAdapter` . . . 495
Class `Triangle2D` . . . 496
Class `RandomTriangles` . . . 497
9.11.4 Implementing `Shape` directly . . . 498
9.12 Turtle graphics class . . . 499
9.12.1 Specification of the class . . . 499
9.12.2 Implementation of the class . . . 500
Class `Turtle2D` . . . 502
9.12.3 Writing turtle graphics programs . . . 505
Class `PentagonSpinner` . . . 506
9.12.4 Recursive turtle graphics programs . . . 508
Class `RecursiveTreeMaker` . . . 508
9.13 Numerical applications of interfaces . . . 509
9.13.1 Displaying a table of values of a function . . . 510
Interface `DoubleFunction` . . . 511
Class `TableMaker` . . . 511
Class `ExpMinusFunction` . . . 512
Class `CosFunction` . . . 512
Class `TableMakerTester` . . . 513
9.13.2 Function iteration . . . 513
Class `FunctionIterator` . . . 514
Class `FunctionIteratorTester` . . . 515
Class `SquareRootIterator` . . . 516
9.14 Review exercises . . . 517
9.15 Programming exercises . . . 518

10 Graphical Interface Design 523
10.1 Introduction . . . 524
10.2 Basic structure of a GUI application . . . 524
10.2.1 Basic template for GUI applications . . . 524
Class `ApplicationTemplate` . . . 524
10.3 GUI components and the greeting application . . . 526

10.3.1 Greeting application design . . . 526
10.3.2 Determining what GUI components are needed . . . 526
10.3.3 Creating the GUI components . . . 527
10.3.4 Choosing a layout manager for the GUI components . . . 528
10.3.5 Adding GUI components to the frame . . . 528
10.3.6 Sending events to listeners . . . 529
10.3.7 Writing event processing code . . . 530
Class `Greeting1` . . . 531
10.3.8 Using a button in the greeting application . . . 532
Class `Greeting2` . . . 533
10.3.9 Multiple types of action event responses . . . 535
Class `Greeting3` . . . 536
10.3.10 Using inner classes to specify event handlers . . . 537
Class `Greeting4` . . . 538
External names for inner classes . . . 540
10.4 Numeric fields and the temperature application . . . 540
10.4.1 Numeric and string conversions . . . 540
Converting numbers to strings . . . 540
Converting strings to numbers . . . 541
Using numeric fields in the temperature application . . . 541
10.4.2 Temperature application . . . 541
Class `Temperature` . . . 541
10.5 Multi-line text fields . . . 543
10.5.1 `JTextArea` objects . . . 543
10.5.2 Investment application . . . 543
Using panels to design GUI layouts . . . 544
Doing the calculations . . . 545
Class `Investment` . . . 546
10.6 Using inheritance to design smarter text fields . . . 548
10.6.1 Structure of the `JTextField` class . . . 548
Class `InputJTextField` . . . 549
10.7 GUI for the loan repayment class . . . 552
10.7.1 GUI version of the loan repayment class . . . 552
Class `LoanRepaymentTableGUI` . . . 553
10.8 Unit conversion application . . . 555
10.8.1 `Conversions` class . . . 558
Class `Conversions` . . . 558
10.9 Inheritance and listener interfaces . . . 560
10.9.1 `ActionListener` interface . . . 561
10.9.2 `WindowListener` interface . . . 561
10.10 Average mark calculator . . . 563
10.10.1 Console version of average mark calculator . . . 563
Class `MarkAverageConsole` . . . 563
10.10.2 GUI version of average mark calculator . . . 564

Class `MarkAverageGUI` . . . 567
10.11 GUI version of the `RandomTriangles` class . . . 569
10.11.1 `ControlPanel` class . . . 569
Class `ControlPanel` . . . 570
Class `RandomTrianglesGUI` . . . 571
10.12 Inheritance and the `GraphicsFrame` class . . . 573
10.12.1 `GraphicsFrame` class . . . 574
Class `GraphicsFrame` . . . 574
10.13 Applets . . . 575
10.13.1 RGB color applet . . . 576
Laying out the components . . . 577
Class `RGBColorApplet` . . . 581
10.13.2 Running applets from `BlueJ` . . . 584
10.13.3 Running Java applets in a browser . . . 586
10.13.4 Launching Java applications from an applet . . . 587
Class `ApplicationLauncher` . . . 588
10.14 Review exercises . . . 590
10.15 Programming exercises . . . 590

11 Files and Streams 593
11.1 Introduction . . . 594
11.2 File concepts . . . 594
11.2.1 What is a file? . . . 594
11.2.2 File organization on a sisk . . . 594
11.2.3 Efficient file access using buffering . . . 595
11.2.4 File access rights . . . 595
11.2.5 Reading and writing sequential files . . . 595
Reading a sequential file . . . 595
Writing a sequential file . . . 596
11.2.6 File access methods . . . 597
Sequential access files . . . 597
Random access files . . . 597
11.3 File structure . . . 598
11.3.1 Text file structure . . . 598
11.3.2 Binary file structure . . . 599
11.3.3 Streams . . . 599
11.4 Java stream and file I/O class hierarchies . . . 600
11.4.1 `InputStream` hierarchy . . . 600
11.4.2 `OutputStream` hierarchy . . . 603
11.4.3 `Reader` hierarchy . . . 604
11.4.4 `Writer` hierarchy . . . 605
11.4.5 `File` class . . . 606
11.4.6 Summary . . . 606
11.5 File I/O error handling using exceptions . . . 607

11.5.1 Unchecked exceptions . . . 607
11.5.2 Checked exceptions . . . 607
11.6 Reading and writing byte streams . . . 608
11.6.1 Reading bytes from an input byte stream . . . 608
11.6.2 Writing bytes to an output byte stream . . . 610
11.6.3 File copy program . . . 611
Class `FileCopier` . . . 611
Class `FileCopyChooser` . . . 613
11.7 Reading and writing character streams . . . 616
11.7.1 Reading characters from an input stream . . . 616
11.7.2 Writing characters to an output stream . . . 618
`PrintWriter` class . . . 619
11.7.3 Simple search program . . . 620
Class `FileSearcher` . . . 620
11.8 Viewing byte contents of files . . . 622
11.8.1 `ByteViewer` class . . . 623
Class `ByteViewer` . . . 623
11.9 Text files of `BankAccount` objects . . . 626
11.9.1 Reading and writing single-line records . . . 626
Implementing the `writeAccount` method . . . 627
Implementing the `readAccount` method . . . 627
11.9.2 Finding a home for `readAccount` and `writeAccount` . . . 629
11.9.3 Extending the `BufferedReader` class . . . 629
Class `BufferedAccountReader` . . . 629
11.9.4 Extending the `PrintWriter` class . . . 632
Class `PrintAccountWriter` . . . 632
11.10 Bank account text file processing . . . 633
11.10.1 Finding the maximum balance among the accounts . . . 635
Class `MaxBalanceCalculator` . . . 636
11.10.2 Account processing . . . 637
11.10.3 Reading database files into arrays . . . 637
11.10.4 The Dynamic `ArrayList<E>` Class . . . 638
Class `ListProcessor` . . . 640
11.11 Binary object files using Polymorphism . . . 643
11.11.1 Writing serialized objects to a file . . . 643
Class `AccountListObjectWriter` . . . 644
11.11.2 Reading serialized objects from a file . . . 646
Class `AccountListObjectReader` . . . 646
Reading more than one object at a time . . . 648
11.12 Review exercises . . . 649
11.13 Exercises . . . 650

12 Searching and Sorting Algorithms 657
12.1 Introduction . . . 658

12.2 Minimum and maximum algorithms . . . 658
12.3 Running time of an algorithm . . . 661
12.4 Searching algorithms . . . 662
12.4.1 Linear search algorithm . . . 662
Order of linear search . . . 663
12.4.2 Recursive binary search algorithm . . . 664
12.4.3 Non-recursive binary search algorithm . . . 665
12.4.4 Running time of binary search algorithm . . . 666
12.4.5 Class of static searching methods . . . 667
Class `IntArraySearch` . . . 668
12.4.6 Testing the search algorithms . . . 669
Class `IntArraySearchTester` . . . 669
12.5 Sorting algorithms . . . 671
12.5.1 Selection sort algorithm . . . 671
12.5.2 Running time for selection sort . . . 673
12.5.3 Insertion sort algorithm . . . 674
12.5.4 Running time for insertion sort . . . 676
12.5.5 Simulation to compare selection and insertion sort . . . 676
Class `QuadraticSortTimer` . . . 677
12.5.6 Mergesort . . . 680
12.5.7 Merge algorithm for two sorted subarrays . . . 680
12.5.8 Running time for mergesort . . . 686
12.5.9 File merge example . . . 686
Class `FileMerger` . . . 688
12.5.10 Quicksort . . . 690
Partitioning an array . . . 690
An implementation of partition . . . 692
12.5.11 Running time for quicksort . . . 693
12.5.12 Simulation to compare mergesort and quicksort . . . 694
Class `FasterSortTimer` . . . 694
12.5.13 Class of static sorting methods . . . 696
Class `IntArraySort` . . . 696
12.5.14 Testing the sorting algorithms . . . 700
Class `SortTester` . . . 701
12.6 Generic object sorting . . . 702
12.6.1 The `Comparator` interface . . . 703
12.6.2 `GenericArraySort` class . . . 705
Class `GenericArraySort` . . . 705
12.6.3 Sorting strings in lexicographical order . . . 709
Class `StringComparator` . . . 709
Class `StringDecreasingComparator` . . . 710
Class `GenericStringSortTester` . . . 710
12.6.4 Comparing `BankAccount` objects . . . 713
Class `AccountNumberComparator` . . . 713

Class `AccountBalanceComparator` . . . 713
Class `BankAccountSortTester` . . . 714
12.7 `Arrays` class . . . 714
12.7.1 `Comparable` interface . . . 715
12.7.2 Searching algorithms . . . 715
12.7.3 Sorting algorithms . . . 716
12.8 Exercises . . . 717

13 Introduction to Data Types and Structures **723**
13.1 Introduction . . . 724
13.2 Abstract data types . . . 724
13.2.1 Classification of ADT operations . . . 724
Create operation . . . 725
Copy operation . . . 725
Destroy operation . . . 725
Modification operations . . . 725
Inquiry operations . . . 725
13.2.2 Pre- and post-conditions . . . 725
13.2.3 Simple ADT examples . . . 726
An integer ADT . . . 726
A floating point ADT . . . 726
A character ADT . . . 727
A boolean ADT . . . 727
13.2.4 Some common structured ADTs . . . 727
The array ADT . . . 727
The string ADT . . . 727
13.2.5 User defined ADT examples . . . 728
A dynamic array ADT . . . 728
A bag ADT . . . 728
13.3 Implementing an ADT . . . 728
13.3.1 Implementation of the `Bag<E>` ADT . . . 729
Designing the `Bag<E>` ADT . . . 729
Interface `Bag<E>` . . . 729
Designing a fixed size implementation . . . 730
Choosing a data structure . . . 731
Implementing the constructors . . . 732
Implementing the methods . . . 732
Class `FixedBag<E>` . . . 733
Converting to a dynamic implementation . . . 735
Class `DynamicBag<E>` . . . 736
13.3.2 Implementation of the `DynamicArray` ADT . . . 737
Designing the `Array` ADT . . . 737
Interface `Array<E>` . . . 738
Designing a dynamic implementation . . . 739

Using the design . . . 739
Implementing the constructors and methods . . . 740
Class `DynamicArray<E>` . . . 740
13.4 Java Collections Framework (JCF) . . . 742
13.4.1 Interface hierarchy . . . 742
13.4.2 Traversing a collection with an iterator . . . 744
13.4.3 `Iterable<E>` interface . . . 746
13.5 `Collection<E>` and `Set<E>` interfaces . . . 746
13.5.1 `Collection<E>` interface . . . 746
13.5.2 `Set<E>` interface . . . 748
Set theory interpretation of the bulk set methods . . . 749
13.6 Set Implementations and examples . . . 749
13.6.1 `HashSet<E>` implementation of `Set<E>` . . . 749
13.6.2 `LinkedHashSet<E>` implementation of `Set<E>` . . . 750
13.6.3 `TreeSet<E>` implementation of `SortedSet<E>` and `Set<E>` . . . 750
13.6.4 Simple set examples . . . 751
13.6.5 Removing duplicates from a list of words . . . 753
Class `RemoveDuplicateWords` . . . 753
13.7 `List<E>` and `ListIterator<E>` interfaces . . . 754
13.7.1 `List<E>` interface . . . 754
13.7.2 `ListIterator<E>` interface . . . 757
13.8 `List<E>` implementations and examples . . . 757
13.8.1 `ArrayList<E>` implementation of `List<E>` . . . 757
13.8.2 `LinkedList<E>` implementation of `List<E>` . . . 758
13.8.3 Simple list examples . . . 758
13.8.4 Book inventory example . . . 760
Class `Book` . . . 760
Class `BookList` . . . 762
13.8.5 Insertion in a sorted list . . . 764
Class `SortedListExample` . . . 766
13.9 Map data type . . . 768
13.9.1 Name-age example . . . 768
13.9.2 Basic map operations . . . 768
13.9.3 Hash tables and codes . . . 769
13.10 The `Map<K,V>` interface . . . 770
13.11 Map implementations and examples . . . 772
13.11.1 `HashMap<K,V>` implementation of `Map<K,V>` . . . 773
13.11.2 `LinkedHashMap<K,V>` implementation of `Map<K,V>` . . . 773
13.11.3 `TreeMap<K,V>` implementation of `Map<K,V>` . . . 774
13.11.4 Simple map examples . . . 774
13.11.5 Hours worked example . . . 777
Class `HoursWorked` . . . 778
13.11.6 Favorites map with maps as values . . . 780
13.12 Recursion examples using maps . . . 783

13.12.1 The Fibonacci sequence 783
Class `Fibonacci` 783
13.12.2 The Q-sequence 784
Class `QSequence` 785
13.13 `Collections` utility class 786
13.13.1 Book list sorting example 786
Class `BookComparator` 787
Class `SortBookList` 787
13.14 Programming exercises 789

List of Figures

1.1 An algorithm for $\sqrt{2}$ . . . 3
1.2 Computer memory locations (bytes) and addresses . . . 5
1.3 Block diagram of a computer system . . . 5
1.4 Mathematical description of addition problem . . . 9
1.5 Java program statements to add three integers . . . 9
1.6 The compilation process . . . 9
1.7 The interpretation process . . . 10
1.8 The Java Virtual Machine . . . 11

2.1 Pictorial representation of uninitialized variables . . . 17
2.2 Pictorial representation of initialized variables . . . 18
2.3 Using BeanShell to display results. . . . 23
2.4 Matching actual and formal arguments . . . 31

3.1 The `chapter3` project . . . 41
3.2 Enter constructor args and invoking methods . . . 42
3.3 After choosing the `getArea` method the result for the area is displayed. . . . 42
3.4 Three `CircleCalculator` objects for radii 2, 3, and 4. . . . 42
3.5 Instance data fields for `circle1` objects in (a) and `circle2` objects in (b). . . . 44
3.6 Triangle formulas given two sides a and b, and the contained angle γ. . . . 46
3.7 Dialog box for entering the arguments to construct a `TriangleCalculator` object 47
3.8 The method menu for a `TriangleCalculator` object. . . . 47
3.9 third side and sum of angles . . . 48
3.10 Inspecting a `TrianglCalculator` object . . . 48
3.11 Dialog box for constructing `QuadraticRootFinder` object . . . 52
3.12 The method menu for a `QuadraticRootFinder` object. . . . 53
3.13 The roots of the quadratic equation for $a = 1$, $b = -2$ and $c = 3/4$. . . . 53
3.14 The dialog box for the `setC` method. . . . 53
3.15 The roots of the quadratic equation for $a = 1$, $b = -2$, $c = 1$. . . . 53
3.16 Inspect for a `QuadraticRootFinder` object . . . 54
3.17 Another example of inspect . . . 58
3.18 Using BeanShell to construct a `CircleCalculator` object and invoke its methods. 61
3.19 An editor window. . . . 70
3.20 Generated Java documentation obtained from the implementation/interface button. 71

3.21 A web page for the chapter3 project documentation 72
3.22 BlueJ syntax error for a missing semi-colon. 73
3.23 Detailed BlueJ error message for a missing semi-colon. 73
3.24 Detailed BlueJ error message for an undeclared variable. 74
3.25 Detailed BlueJ error message for a variable declared twice. 74
3.26 Detailed BlueJ error message for misspelled constructor name. 75
3.27 A template for a simple Java class declaration. 81
3.28 A template for a simple Java constructor declaration. 81
3.29 A template for a simple Java method declaration. 82

4.1 A string object in memory and a reference to it called `greeting` 93
4.2 BlueJ terminal window 101
4.3 Matching actual and formal constructor arguments 105
4.4 (a) Constructing a `BankAccount` object, (b) the object menu 109
4.5 (a) Association in BlueJ using dashed line, (b) output for a test case. 119
4.6 Aggregation with the `Point` and `Circle` classes. 125
4.7 Using inspect for object references 126
4.8 Getting an object on the workbench 126
4.9 Testing `CalendarMonth` in BlueJ 137
4.10 Box and arrow representation of an object and a reference to it. 144
4.11 Comparison of primitive and reference types. 145
4.12 a = b for primitive types 146
4.13 a = b for reference types 147
4.14 A side-effect after `p` changes the `x` from `3` to `999` 154
4.15 The side-effect disappears after `p` changes `x` from `3` to `999`. 158
4.16 The `TransferAgent` example in BlueJ 160
4.17 Correspondence between call expressions and prototypes 161
4.18 Call by value using references. 168
4.19 The `main-method` project. 170

5.1 An empty drawing window 183
5.2 User and device space 187
5.3 Drawing a line 190
5.4 The biggest rectangle 191
5.5 The biggest rectangle after resizing window 192
5.6 Inheritance hierarchy and `Shape` interface diagram for graphics classes 195
5.7 Ellipse, circle, and frames 198
5.8 Drawing round rectangles 200
5.9 The three types of arcs 201
5.10 Anti-aliased version of `DrawArc` 203
5.11 `DrawArc2` showing colors and stroke thickness 2 206
5.12 Designing the happy face with boxes 207
5.13 `FaceMaker1`: a no-frills happy face 210
5.14 `FaceMaker2`: a colorful happy face 212

5.15 `FaceMaker3`: translating and scaling the happy face . . . 215
5.16 `FaceMaker4`: a half-size happy face . . . 218
5.17 `FaceMaker5`: a rotated happy face . . . 218
5.18 `FaceMaker6`: four happy faces . . . 219
5.19 Bar graph coordinate transformation . . . 226
5.20 Output of the `BarGraph1` program . . . 230
5.21 World to default user coordinate transformation . . . 230
5.22 Output of the `DrawPentagon1` class . . . 237
5.23 World to default user coordinate transformation . . . 238
5.24 A colorful happy face with ears . . . 247
5.25 A fish . . . 248
5.26 Winnie the Pooh . . . 248
5.27 Nakami Araki . . . 249
5.28 Centered coordinate system . . . 250

6.1 A template for the if-statement . . . 254
6.2 A flowchart for the execution of an if-statement . . . 255
6.3 Flowchart for an if-statement with no else-part . . . 259
6.4 Template for the multiple if-statement . . . 264
6.5 A flowchart for the execution of a multiple if-statement . . . 266
6.6 forgetting the braces . . . 266
6.7 Pseudo-code algorithm for the leap year problem . . . 272
6.8 Pseudo-code algorithm for roots of a cubic equation . . . 310

7.1 A template for the while-statement . . . 314
7.2 A flowchart for the execution of a while-statement . . . 315
7.3 Algorithm to convert a string to an integer . . . 318
7.4 Algorithm to compute $\sqrt{a}$. . . . 320
7.5 Pseudo-code algorithm for doubling your money . . . 322
7.6 Pseudo-code factorization algorithm . . . 324
7.7 A template for the do-while statement . . . 332
7.8 A flowchart for the execution of a do-while statement . . . 332
7.9 A template for a general loop structure. . . . 333
7.10 A template for the for-statement. . . . 335
7.11 A flowchart for the execution of a for-statement . . . 335
7.12 A for-loop expressed as a while-loop . . . 343
7.13 Pseudo-code loan repayment algorithm . . . 344
7.14 5 rows of 10 circles touching each other . . . 350
7.15 Approximating part of a function with a line segment . . . 356
7.16 Pseudo-code graph drawing algorithm . . . 357
7.17 `SineGraph` output window . . . 359
7.18 Partial sum algorithm for e . . . 370
7.19 Pseudo-code non-recursive gcd algorithm . . . 373
7.20 A pseudo-code gcd algorithm using subtraction . . . 374

7.21 Output from the GridMaker class . . . 375
7.22 ConcentricCircles output window . . . 376

8.1 Constructing an array of integers . . . 383
8.2 Constructing an array, b, of three BankAccount objects . . . 388
8.3 Pseudo-code algorithm for maximum array element . . . 397
8.4 Pseudo-code linear search algorithm . . . 400
8.5 Pseudo-code bubble sort algorithm . . . 403
8.6 Pseudo-code polynomial evaluation algorithm . . . 407
8.7 Line graph for vertex array $\langle (x_0,y_0),(x_1,y_1),\ldots,(x_6,y_6)\rangle$ . . . 412
8.8 Line graph from SimpleTester . . . 418
8.9 A RandomTester output window . . . 420
8.10 A two-dimensional array with different length rows . . . 425
8.11 Output of the PentagonalStar program . . . 447
8.12 A duck using arrays of points . . . 448

9.1 Part of the domestic animal inheritance hierarchy . . . 454
9.2 Bank account inheritance hierarchy . . . 454
9.3 Employee inheritance hierarchy . . . 455
9.4 A template for a simple Java subclass declaration. . . . 455
9.5 Part of the GUI component hierarchy . . . 457
9.6 Inheritance in BlueJ is indicated by a solid arrow . . . 460
9.7 (a) CircleCalculatorA object menu, (b) CircleCalculatorB object menu . . . 461
9.8 Bank account inheritance project . . . 465
9.9 BankAccount inheritance . . . 466
9.10 Output of ShapeTester class . . . 489
9.11 Polar coordinates of a point (x_1,y_1) . . . 501
9.12 Output for PentagonSpinner program. . . . 507
9.13 Output window for the RecursiveTreeMaker class . . . 510
9.14 Another possible Employee inheritance hierarchy . . . 519

10.1 Greeting application before Enter key is pressed . . . 527
10.2 Greeting application after Enter key is pressed . . . 527
10.3 Greeting application illustrating the flow layout . . . 528
10.4 When an event occurs the component calls the appropriate event handler . . . 529
10.5 Adding listeners (event handlers) to a GUI component . . . 530
10.6 Greeting application containing a button . . . 533
10.7 Greeting application containing an exit button . . . 535
10.8 The temperature conversion application frame . . . 540
10.9 The investment application . . . 544
10.10 LoanRepaymentTableGUI application . . . 556
10.11 Conversions application . . . 557
10.12 MarkAverage window for one set of marks . . . 564
10.13 MarkAverage window for two sets of marks . . . 565

10.14 The random triangles GUI using a `ControlPanel` and a `JPanel` 569
10.15 `RGBColorApplet` in the applet viewer window 577
10.16 `RGBColorApplet` as a GUI application 578
10.17 `RGBColorApplet` in a BlueJ project 585
10.18 `RGBColorApplet` class menu 585
10.19 Defining `RGBColorApplet` applet parameters 585
10.20 `RGBColorApplet` running in Internet Explorer 587
10.21 `ApplicationLauncher` running in Internet Explorer 588
10.22 `InvestmentTable` application 591
10.23 `ChangeMaker` application 591

11.1 The `InputStream` and `OutputStream` hierarchies 601
11.2 The `Reader` and `Writer` hierarchies 602
11.3 Part of the unchecked exception sub-hierarchy of `RunTimeException`. 607
11.4 Part of the checked exception sub-hierarchy of `IOException`. 608
11.5 The `ByteViewer` GUI showing `ByteViewer.class` 622
11.6 Pseudo-code transaction processing algorithm 656

12.1 Pseudo-code algorithm for minimum array element 659
12.2 Pseudo-code linear search algorithm 663
12.3 Alternate Pseudo-code linear search algorithm 663
12.4 Pseudo-code recursive binary search algorithm 666
12.5 Pseudo-code non-recursive binary search algorithm 667
12.6 Pseudo-code selection sort algorithm 672
12.7 An insertion sort example for $\langle 44,55,12,42,94,18,6,67\rangle$. 675
12.8 Pseudo-code insertion sort algorithm 675
12.9 Mergesort example for the array $\langle 8,1,6,4,10,5,3,2,22\rangle$ 681
12.10 Merging the subarrays $\langle 1,8,12,15,17\rangle$ and $\langle 2,9,10,19,21,23,25\rangle$ 682
12.11 Pseudo-code merge algorithm for two subarrays 683
12.12 Partitioning subarrays for quicksort 691
12.13 Partitioning $\langle 2,1,6,4,8,5,3,10\rangle$ into subarrays $\langle 3,1,2,4\rangle$ and $\langle 8,5,6,10\rangle$. 693

13.1 JCF related interface hierarchy 742
13.2 The `Collection<E>` interface 743
13.3 The `Iterable<E>` interface 744
13.4 The `Iterator<E>` interface 744
13.5 The `Set<E>` interface 748
13.6 The `HashSet<E>` class 750
13.7 The `LinkedHashSet<E>` class 750
13.8 The `TreeSet<E>` class 751
13.9 The `List<E>` interface 755
13.10 The `ListIterator<E>` interface 755
13.11 Indices for a list iterator 757
13.12 The `ArrayList<E>` class 758

13.13 The `LinkedList<E>` class 759
13.14 A two-column representation of the name-age map 768
13.15 A simple hash table of size 11 using $h(k) = k \bmod 11$. 769
13.16 `Map` interface 771
13.17 The `HashMap<K,V>` class 773
13.18 The `LinkedHashMap<K,V>` class 773
13.19 The `TreeMap<K,V>` class 774
13.20 A map of maps 781
13.21 Memory game algorithm 792

List of Tables

1.1 Intel 8088 machine language program and data 7
1.2 Intel 8088 assembly language instructions 8
1.3 Example of Java source code to bytecode translation 12

2.1 The Java integer and floating point primitive types. 16
2.2 Translation of mathematical expressions to Java. 21

3.1 BlueJ actions and their corresponding Java statements. 59

5.1 The standard colors 204

6.1 The six comparison operators 253
6.2 The three basic logical operators 268
6.3 Truth tables for AND, OR, and NOT 269
6.4 ASCII codes for characters 276
6.5 Lexicographical string comparison using compareTo 277
6.6 Table of outcomes for the paper, scissors, rock game 289
6.7 Truth table for the exclusive or 303

7.1 A loan repayment table 344

8.1 Bubble sort example 403
8.2 A deck of playing cards 433

12.1 Binary search example 664
12.2 Comparison of n and $\log_2 n$ 665
12.3 Low and high indices for non-recursive binary search 666
12.4 Selection sort example 672
12.5 Counting inner loop executions for selection sort 674
12.6 Counting inner loop executions for worst case insertion sort 676
12.7 Comparing average times for selection sort and insertion sort. 679
12.8 Growth rates for $\log_2 n$, n, $n \log_2 n$, and n^2. 687
12.9 Comparing average times for mergesort and quicksort. 696

Chapter 1

Introduction to Computation

Algorithms, Processors, and Programs

Outline

Concepts of algorithm, processor, and memory

Translation and interpretation of programs

Java virtual machine (JVM)

Java source code compiler and byte code

1.1 Introduction

In this chapter we present a brief overview of the nature of computation and the process whereby a computer program written in a high-level language such as Java can be executed by a computer system. First we introduce the concept of an algorithm which describes how to solve a problem or accomplish a task in a sequence of well defined steps. We are interested in algorithms that can be executed by a computer system. To do this it is necessary to translate the algorithm into a program in a language that the computer understands. When this is done the computer can execute the steps of the program to complete the task for which the algorithm was designed.

The design of efficient algorithms, their expression in a programming language, and their execution by a computer system is the essence of computer science. We are primarily concerned with the problem solving process whereby algorithms are constructed, converted to programs in the Java language, compiled, and executed by the Java Virtual Machine.

1.2 Algorithms

We are all somewhat familiar with algorithms. A recipe for baking a loaf of bread is a standard example. The recipe is expressed as a number of steps that, if followed, will produce a loaf of bread. It is important that each step be specified without ambiguity. For example, the step "add some flour" is not precise enough and should be replaced by something like "add 3 cups of flour". Here we are assuming that the baker knows how much is in a cup. There are many other examples from everyday life that can be expressed as a number of English statements that are sufficiently detailed for an average person to understand and follow.

A recipe is not unique. It is possible to express a recipe for a particular kind of bread as a large number of simple steps for a novice baker or as a smaller number of steps for an experienced baker, who would understand a vague statement such as "season to taste". A novice would require further instructions on the kind and quantity of spices or herbs to add. Generalizing, we can say that an **algorithm** is a sequence of unambiguous steps, for accomplishing some task or process. Each step is a simple instruction or statement. The word "sequence" is important since it implies that the steps must be performed in a specific order.

We are interested in algorithms that can eventually be executed by a computer system so it will be very important that the individual instructions be unambiguous. We also require that the total number of steps in an algorithm is finite although algorithms can have an infinite number of steps and still be well-defined. For example, in mathematics any algorithm to calculate $\sqrt{2}$ necessarily has an infinite number of steps since the answer is an irrational number that has an infinite non-repeating decimal expansion.

We can rephrase the algorithm in a finite form by asking for an algorithm to compute $\sqrt{2}$ accurate to a specified number of digits. Such an algorithm is given in Figure 1.1. Here the left arrow means that the value on the right side of the arrow is computed and assigned as the value of the variable on the left side of the arrow. This algorithm is still not completely precise because it doesn't specify how many digits to use in the intermediate calculations. For example, if you use at least 16 significant digits in each arithmetic operation then the algorithm will produce the approximation 1.41421356237 accurate to the number of digits shown. The algorithm also assumes that

```
**ALGORITHM**: SquareRootOfTwo
$x_0 \leftarrow 1$
$x_1 \leftarrow (x_0 + 2/x_0)/2$
$x_2 \leftarrow (x_1 + 2/x_1)/2$
$x_3 \leftarrow (x_2 + 2/x_2)/2$
$x_4 \leftarrow (x_3 + 2/x_3)/2$
$x_5 \leftarrow (x_4 + 2/x_4)/2$
**RETURN** $x_5$
```

Figure 1.1: An algorithm for computing an approximation to $\sqrt{2}$

the user knows how to add and divide real numbers (floating point numbers). Otherwise it would have to be rewritten as an enormous number of smaller steps that explain the $+$ and $/$ operations.

Thus, algorithms have a context: they assume that the user (or computer) understands how to execute a basic set of unambiguous instructions. Unfortunately, as we will see, the instructions understood by a computer system are very elementary and far removed from the combination of natural language and mathematics that are normally used to formulate algorithms at the human problem solving level.

1.3 Processors

A processor is any device, computer, calculator, or human being, for example, that can process or execute a sequence of instructions. A computer system contains a processor, called a central processing unit or CPU, which knows how to perform (execute) instructions from a specific instruction set called the machine language of the CPU. Each kind of CPU has a different machine language. For example, the machine language of the Pentium CPU used in a PC is very different from that of the PowerPC CPU used in the Macintosh computer. The CPU in one of these personal computers is built on a single microchip and is called a microprocessor.

Each machine language instruction is defined by a specific binary code: a binary string of 0's and 1's. This code determines what operation is to be performed and where to locate any required data. This data is also represented in binary form. Compared to human problem solving instructions and algorithms, written in English and mathematics, machine language instructions are very primitive. In fact a single English instruction such as "compute the value of this polynomial at $x = 3.14$" may have to be translated into several or even hundreds of machine language instructions before it is understood by the CPU.

Fortunately, this translation process can be automated today, although this wasn't always the case for the first electronic computers. Our goal in this chapter is to briefly explain how the human problem solving domain is mapped down to this primitive machine language level.

1.3.1 Functional units of a CPU

A CPU is composed of three basic functional units.

- **Arithmetic-Logic Unit (ALU):** performs logical and arithmetic operations on data, such as adding, subtracting, multiplying or dividing two binary numbers, determining if a number is zero or greater than zero, or comparing two numbers to see if one is equal to the other, less than the other, or greater than the other.

- **Registers:** storage locations within the CPU that hold data and numbers needed by the ALU in performing operations, storing the results of these operations, or storing the address of the next instruction to be executed.

- **Control Unit (CU):** controls the operation of the processor such as fetching from memory the next instruction to be executed and any data needed, or decoding binary instructions to determine what operation is to be performed.

1.4 Memory and I/O devices

To execute a machine language program the CPU needs to access the instructions and data. In the early computer systems instructions and data were separate concepts. Data was stored in a memory but the program was not stored anywhere. Instead it was painstakingly constructed by connecting various cables and hardware together. In essence the control unit was being "re-programmed" for each program.

Von Neumann introduced what is known as the "stored-program concept" that is the basis of modern computing systems. The idea is that since instructions and data are both strings of binary digits (bits) they can both be stored in a memory external to the CPU: instructions are just a form of data understood by the CPU. Thus, a machine language program and its data is simply a large string of 0's and 1's stored in a memory. The CPU keeps track of the location in memory of the next instruction to be executed and each instruction specifies the location in memory of any required data. To execute another program it is not necessary to "rewire" the hardware: simply store a new program and its data in the memory and tell the CPU where to find the first instruction.

A memory location is a basic unit of memory that has an address associated with it to distinguish it from other memory locations. Today most memories are organized as a sequence of bytes. Each byte is a memory location containing 8-bits of information. We say that the memory is "byte-addressable" since every byte has a unique address. Each instruction is now stored in one or more bytes of memory and similarly for data items such as integers and real numbers. Many modern microprocessors use 32-bit instructions each stored in four consecutive bytes and use 32-bit addresses to locate each byte. Similarly, they can process data in arithmetic and logical operations using up to 32-bits at a time. A pictorial representation of six consecutive bytes of memory, their addresses, and their content is shown in Figure 1.2. The content of each of the six bytes is shown inside the boxes but the addresses are shown as labels beside each box. They are obtained using address decoding logic built into the memory chips and CPU.

There are two categories of memory chips. ROM is read only memory that once written can not be changed and RAM is memory that can be read or written. The contents of RAM memory

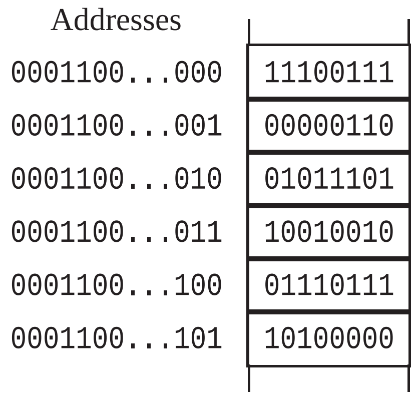


Figure 1.2: Computer memory locations (bytes) and addresses

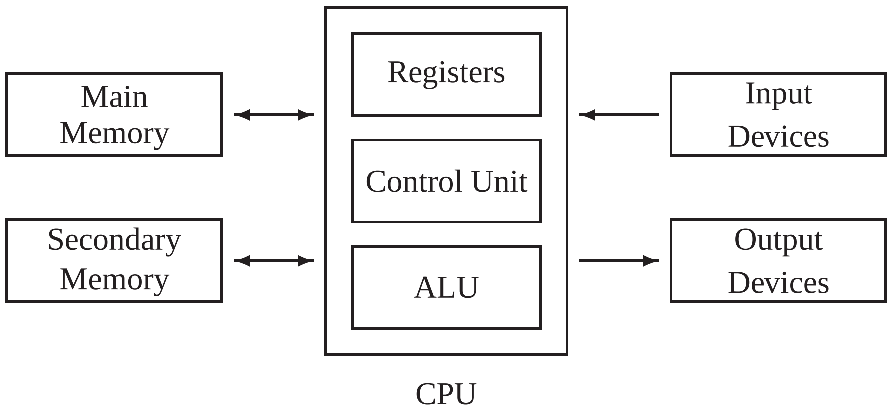


Figure 1.3: Block diagram of a computer system

(volatile memory) are lost when power to the memory chips is interrupted (by turning the computer off, for example) but the contents of ROM memory is permanent. There are also intermediate types of memory that can be read and written but do not lose their contents when power is interrupted.

The ROM and RAM memory devices are often called main memory as opposed to secondary memory which consists of external storage devices such as floppy disks and hard disks. They are needed to store files and results of computations in a more permanent form that is not lost when the computer is turned off.

Floppy disks and hard disks are both input and output devices, whereas the keyboard and the mouse are classified as input devices. Here we use the term “input data” to describe incoming data that is needed by the processor as it is executing instructions. The term “output data” refers to data produced by the processor as a result of executing instructions. The block diagram in Figure 1.3 shows how the CPU communicates with the “outside world” using main memory (RAM and ROM), secondary memory (floppy disks, hard disks), input devices (such as a mouse or keyboard), and output devices (such as a display screen or a printer).

1.5 Programs

A **program** is a representation of an algorithm as a sequence of instructions that can be understood and executed by a processor to complete the task, problem, or process described by the algorithm. For a CPU each program is a sequence of machine language instructions. Since there are only a few registers (32, for example) within a CPU for holding instructions to be executed, the data required, and intermediate results, machine language programs and their data are stored in main memory (see Figure 1.3).

The CPU's control unit (CU) is designed to fetch instructions from memory and have them executed using the ALU for operations and the registers for temporary storage. One of the registers is a special one called the instruction address register (IAR). It's purpose is to hold the memory address of the next machine language instruction to be executed. Program execution at the machine level can be briefly described by the following steps:

1. Load the IAR with the starting address in memory of the first machine language instruction.

2. Use IAR to fetch from memory the instruction to be executed and store it in an instruction register (IR).

3. Use the CU to decode this instruction to determine the specific operation and the location of any data required.

4. Fetch any data required and store it in registers.

5. Execute the instruction using this data.

6. Increment the address in the IAR to the next machine language instruction.

7. Return to step 2.

This fetch-decode-execute cycle is fundamental to the operation of all CPU's and is called interpretation. Thus, a CPU is an interpreter for a machine language program.

1.6 Computer languages

At the lowest level we can write programs directly in machine language as long strings of 0's and 1's. The only advantage is that the CPU directly understands this language so maximum efficiency and execution speed can be achieved. However, no one does this anymore because it is too difficult for all but the simplest programs.

The next step up is to write programs in assembly language. An assembly language is essentially a mnemonic form of machine language. Instructions have names and addresses can be represented symbolically with names rather than explicit 32-bit binary numbers. The CPU doesn't understand assembly language so it is necessary to convert an assembly language program to a

Instructions (hex)	**Instructions (binary)**	
`A1 0000`	`1010 0001`	`0000 0000 0000 0000`
`03 06 0002`	`0000 0011 0000 0110`	`0000 0000 0000 0010`
`03 06 0004`	`0000 0011 0000 0110`	`0000 0000 0000 0100`
`A3 0006`	`1010 0011`	`0000 0000 0000 0110`
Data (hex)	**Data (binary)**	
`00 06`	`0000 0000 0000 0110`	
`00 07`	`0000 0000 0000 0111`	
`01 FF`	`0000 0001 1111 1111`	
`02 0C`	`0000 0010 0000 1100`	

Table 1.1: Intel 8088 machine language program and data in both hexadecimal and binary notation

machine language program. This process of converting a program in one language to a program in another language is called **translation** or **compilation**. Of course, it is necessary to have a machine language program to do the translation. The translator that converts an assembly language program to a machine language program is called an **assembler**. Once this translation process is completed the resulting machine language program can be stored in memory and executed by the CPU.

To briefly illustrate these ideas we will use a machine language program for an Intel 8088 processor (processor used in the original IBM PC) that simply adds three integers.

1.6.1 Machine and assembly languages

To understand what machine and assembly language programs looks like let us consider a small Intel 8088 program to add three numbers and store the results. For this processor the standard size for an integer is 16 bits. This means that each integer occupies two consecutive bytes of memory. Therefore we assume that the three numbers to be added are stored in 6 bytes of memory and 2 further bytes of memory are reserved for storing the sum. Therefore the data part of the program occupies 8 bytes. The processor has an instruction to move a 16-bit integer from memory into a processor register, an instruction to add an integer in memory to a number in the register, and an instruction to move the number in a register back to memory again. Therefore four machine language instructions are required to add the three numbers and store the result. The four machine language instructions and the data are shown in Table 1.1. The right column shows the binary format and the left column shows the more compact hexadecimal notation (in base 10 the 10 digits ‘0’ to ‘9’ are used, in base 2 the 2 digits ’0’ and ’1’ are used, and in base 16 ’0’ to ’9’ and the six new “digits” ’A’ to ’F’ are used).

The first instruction, `A1 0000`, occupies 3 bytes. The first byte, `A1`, is called the opcode and means that this is an instruction that moves a 16-bit number into a specific processor register called `ax`. The remaining two bytes contain a special address used to locate the 16-bit number in a special area of memory called the data segment. The second and third instructions are addition instructions and the fourth instruction is another move instruction that stores the result in the data segment.

The final four rows of the table show the 6 bytes in the data segment containing the three

Instruction	**Destination**	**Source**
`mov`	`ax,`	`i`
`add`	`ax,`	`j`
`add`	`ax,`	`k`
`mov`	`sum,`	`ax`
Variable name	**Storage directive**	**Value**
`i`	`dw`	`6`
`j`	`dw`	`7`
`k`	`dw`	`511`
`sum`	`dw`	`0`

Table 1.2: Intel 8088 assembly language instructions

numbers to be added and the 2 bytes reserved for the answer. In the particular case shown the sum, using decimal notation is `6 + 7 + 511`, and the result (`524` decimal, `020C` hexadecimal) is show in the last two bytes. Thus, the four-instruction program is just the string of 112 bits:

```
1010 0001 0000 0000 0000 0000 0000 0011 0000 0110 0000 0000 0000 0010
0000 0011 0000 0110 0000 0000 0000 0100 1010 0011 0000 0000 0000 0110
```

and is not very intelligible. Nevertheless, this is the only program understood by the processor.

In order to make programming at this level manageable assembly languages were invented. In an assembly language symbolic names are given to instruction opcodes, to processor registers, and to memory addresses. For example, our four instruction machine language program in Table 1.1 is shown in assembly language in Table 1.2. For example, the instruction

```
mov ax, i
```

corresponds to the first three bytes (`A1 0000`) of the machine language program. Instructions that move data from one place to another have the name `mov` and one of the 16-bit processor registers has the name `ax`. Also we have used the symbolic names `i`, `j`, `k` for the addresses of the three integers to be added, and the name `sum` for the address of the memory location that will hold the result. These symbolic names are used as labels on `dw` directives (`dw` means "define word") which allocate memory for data in 16-bit units. After the statements are executed by the processor the result (`524` decimal, `020C` hex) will be stored in the two bytes reserved for `sum` in the last row of Table 1.2. We don't care about the actual addresses: the assembler will take care of it. It is clear that this program is much more understandable than the 112-bit string of 0's and 1's in machine language.

1.6.2 High-level languages

Even though an assembly language program is much easier to write and read than a machine language program, it is still very far from the familiar mathematical notation used at the human-level of problem solving which we might express as shown in Figure 1.4.

$$\begin{array}{ll} i \leftarrow 6 & \text{(assign 6 to } i\text{)} \\ j \leftarrow 7 & \text{(assign 7 to } j\text{)} \\ k \leftarrow 511 & \text{(assign 511 to } k\text{)} \\ sum \leftarrow i+j+k & \text{(assign their sum to } sum\text{)} \end{array}$$

Figure 1.4: Mathematical description of addition problem

```
int i = 6;
int j = 7;
int k = 511;
int sum = i + j + k;
```

Figure 1.5: Java program statements to add three integers

Therefore, most programs today are written in a high-level language such as C, C++, or Java. For example, the statements in Figure 1.5 show that Java statements that correspond to the machine language program in Table 1.1 or the assembly language program in Table 1.2. Even with no programming experience these statements can probably be understood. The `int` modifier indicates that the name of the following variable refers to an integer number (a 32-bit integer in Java) and the equal sign indicates that the variable on the left of the equal sign receives the value of the expression on the right side of the equal sign. The equal sign corresponds to the left arrow that we have used to denote assignment of a value to a variable.

1.7 Translation and interpretation of programs

Since the CPU only understands machine language it is necessary to convert programs written in assembly language or a high-level language to machine language. There are three ways to do this

1. Write a program that converts the source program to a machine language program for a specific CPU and then have the CPU execute this program directly. This is called translation. The translation of an assembly language program to a machine language program is done by an assembler. In general a program that converts a program in one language to a program in another language is called a **compiler**. This process is illustrated in Figure 1.6.

2. Write a program called an **interpreter** that reads the source program one statement at a time

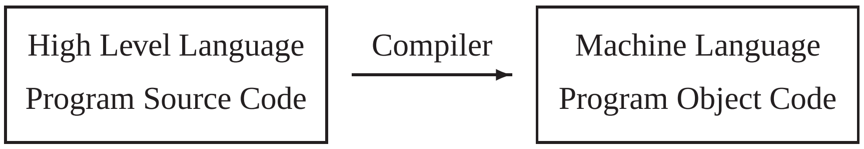


Figure 1.6: The compilation process

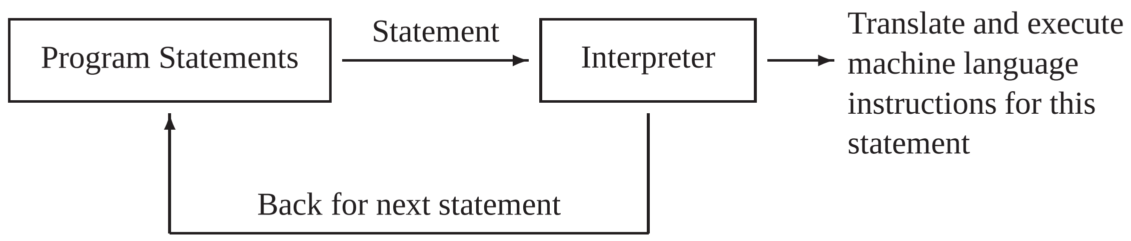

Program Statements
Statement
Interpreter
Translate and execute machine language instructions for this statement
Back for next statement

Figure 1.7: The interpretation process

and has each statement executed by the CPU. Early versions of the BASIC programming language were executed in this way. This interpretation process is shown in Figure 1.7.

3. Write a compiler program that translates the high-level language source program to a program in some intermediate machine language representation for a hypothetical machine called a **virtual machine**. Now write an interpreter for this machine that has the program executed on a real CPU. The interpreter can also compile groups of statements so that, if they are encountered again, they do not need to be re-interpreted again.

Interpreters are more flexible than compilers but they can result in much slower execution since each high-level language statement must be converted "on the fly" each time the statement is encountered. In program loops, where the same sequence of statements can be executed many times, execution will be slower than for the corresponding machine language program for the real CPU. The benefits of both interpretation and compilation can be achieved by the hybrid approach (item 3 above) which is often called "just-in-time" compilation. Most Java program language interpreters can use this hybrid approach to achieve speeds approaching that of the machine language for the CPU.

1.8 Java virtual machine

Java is a peculiar high level language since it is first compiled, not to the machine language of a specific real computer, but rather to a machine neutral object code called **bytecode**. This is illustrated in part (a) of Figure 1.8. The bytecode is the machine language for a hypothetical (virtual) machine called the **Java Virtual Machine** (or JVM). The Java bytecode and the Java compiler are both computer independent. Once a bytecode program is obtained it can be run on any JVM. The JVM is the only machine dependent part of the process, shown in Figure 1.8(b). Here the JVM interprets the bytecode instructions, converts them to the machine language of the real machine, and has them executed on the real machine. Essentially, it is the Java interpreter.

Why does the process of running Java programs require an extra step? While it is possible to design compilers or interpreters to convert Java to a specific machine language, converting it first to Java bytecode has the advantage of portability. A bytecode program does not depend on the hardware of any real computer system. It can be run on any computer system without the need for recompilation as long as someone has written a JVM for the target computer system. Today a JVM is available for almost every computer system.

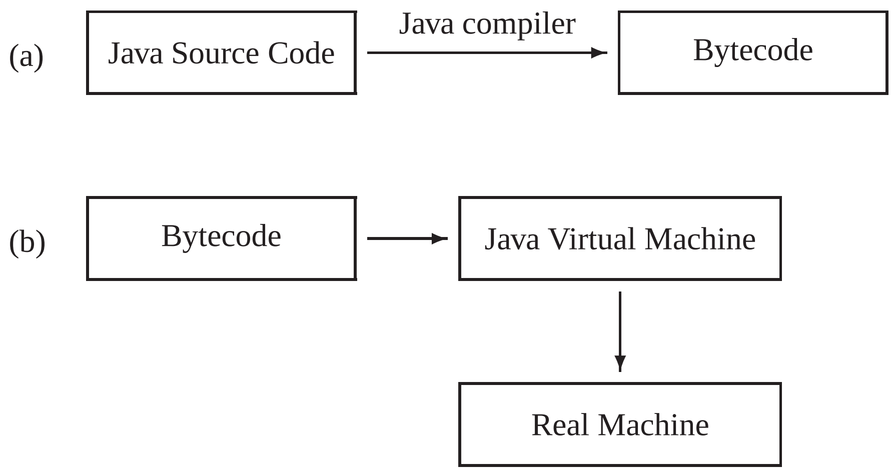


Figure 1.8: The Java Virtual Machine

1.9 Java source code to bytecode translation example

The entire process of translating a program from source code to assembly language to object code can be illustrated by the Java statements in Figure 1.5 that define three integer variables, `i`, `j`, and `k` having the values `6`, `7`, and `511`, respectively, then add them together and store the result in a variable called `sum`.

The Java compiler translates these statements into the language of the Java Virtual Machine. Table 1.3 shows three forms of this translation. The first column shows the 12 bytecode instructions in assembly language for the four Java statements. They are not as recognizable as the Java statements, and are quite different from the Intel 8088 instructions in Table 1.2, but you can still see familiar words such as “load”, “store”, and “add”. The `bipush` instructions store the two byte size integers, `6` and `7`, in an area of memory called the stack, the `sipush` instruction does the same for the 16-bit size integer `511`, the `istore` instructions store data from the stack into memory, and the `iload` instructions load the stack with the numbers to be added. Finally, the `iadd` instructions add the numbers and the result is stored in memory.

The second column shows the bytecode in hexadecimal format, and the final column shows the bytecode in binary form. Thus, the 3 Java statements ultimately result in the following sequence of 0’s and 1’s:

```
0001 0000 0000 0110 0011 1100 0001 0000 0000 0111 0011
1101 0001 0001 0000 0001 1111 1111 0011 1110 0001 1011
0001 1100 0110 0000 0001 1101 0110 0000 0011 0110 0000 0100
```

This program is actually longer than required since it was obtained from the output of the Java compiler which is assuming that local variables are used for the variables `i`, `j`, `k`, and `sum`. If we were to write directly in assembly language the program could be expressed more simply as

```
bipush 6
bipush 7
iadd
```

Assembly Language	**Bytecode (hex)**	**Bytecode (binary)**
`bipush 6`	`10 06`	`0001 0000 0000 0110`
`istore_1`	`3C`	`0011 1100`
`bipush 7`	`10 07`	`0001 0000 0000 0111`
`istore_2`	`3D`	`0011 1101`
`sipush 511`	`11 01 FF`	`0001 0001 0000 0001 1111 1111`
`istore_3`	`3E`	`0011 1110`
`iload_1`	`1B`	`0001 1011`
`iload_2`	`1C`	`0001 1100`
`iadd`	`60`	`0110 0000`
`iload_3`	`1D`	`0001 1101`
`iadd`	`60`	`0110 0000`
`istore 4`	`36 04`	`0011 0110 0000 0100`

Table 1.3: Example of Java source code to bytecode translation

```
sipush 511
iadd
```

which uses the stack area of memory to add three numbers and store the result.

1.10 Review exercises

► **Exercise 1.1** Think of something that you know how to do and try to write an algorithm in English that could be followed by someone who has never done it before.

► **Exercise 1.2** Use your calculator to execute "by hand" the algorithm in Figure 1.1. Compare your answer with the one produced using the square root key.

► **Exercise 1.3** How would you modify the algorithm in Figure 1.1 to compute an approximation to $\sqrt{a}$ for any $a > 0$? Test your algorithm by computing approximations to $\sqrt{3}$, $\sqrt{4}$, $\sqrt{100}$, and $\sqrt{10000}$. Note: a good way to test algorithms is to try them using data for which you know the correct answer (e.g., $\sqrt{10000}$ is 100).

Chapter 2

Fundamental Data Types

Using BeanShell

Outline

Fundamental data types and variables

Declaration and initialization of variables

Arithmetic operations and expressions

Assignment statements

Arithmetic functions from the `Math` class

Using BeanShell to understand basic concepts

2.1 Fundamental data types and variables

Data comes in many types. There are numeric types for integers, characters and floating point numbers, and there are non-numeric types, such as the boolean type which represents the logical values true and false. These fundamental types are called **primitive types**. There are also **object types**, such as the string type for representing strings of characters. We can also define our own types. However, in this chapter we concentrate on the fundamental numeric data types.

Formally, a **data type** has two parts

- A set of values
- A set of operations on these values

In mathematics the most fundamental kinds of data are the integers and real numbers. We will start with these familiar types and see how they are represented as primitive types in Java.

2.1.1 Integer and floating point data types

In mathematics we define $\mathbf{Z}$ to be the set of all integers, and we can define subsets such as $\mathbf{N} = \{n : n \in \mathbf{Z}, n \geq 0\}$ containing only the non-negative integers (read this as the set of all n such that n belongs to $\mathbf{Z}$ and n is greater than or equal to zero). From the integers we can then obtain the set of rational numbers (fractions), $\mathbf{Q} = \{q : q = \frac{a}{b}, a, b \in \mathbf{Z}, b \neq 0\}$. Finally, from the rationals we can obtain the real numbers $\mathbf{R}$.

In algebra, variables can be defined with values taken from some subset of the integers, rationals, or real numbers. Then, following the rules of algebra, operations on these values and variables can be defined.

You should be familiar with the standard operations of addition, subtraction, multiplication, and division, and with the rules of algebra for writing algebraic expressions involving variables, values, and operations. If a and b are integer or real variables we use $a+b$, $a-b$, ab, and a/b to represent these operations. In mathematics variables are normally one letter symbols so the multiplication of a and b is implied by juxtaposition of the variables as in ab. We can also use $a \cdot b$ or $a \times b$ to denote multiplication and this is more appropriate for example in pseudo-code algorithms where variables often have multi letter names. In languages like Java $*$ is used to denote multiplication.

Division requires some care, since a/b is undefined if $b = 0$. Also, if a and b are real numbers then a/b is a real number, but if a and b are integers then a/b need not be an integer. This means there are two kinds of division: real number division and integer division to produce a quotient and remainder. For integer division we can define the quotient and remainder using the

> **Quotient-remainder theorem (Q-R theorem)** If n (the numerator) and d (the divisor or denominator) are non-negative integers and $d \neq 0$, there are unique integers q, called the quotient, and r, called the remainder, such that
>
> $$n = d \cdot q + r, \quad \text{where } r \text{ satisfies } 0 \leq r < d$$

In mathematics the quotient q is defined as the integer division n **div** d, and the remainder r is defined, using the modulus operator, as n **mod** d. Do not confuse the integer division n **div** d with the fraction $\frac{n}{d}$ which is a rational number, or the real number a/b which in general is not an integer. For example 27 **div** 5 is 5 (remainder is 2) and $27/5$ is the real number 5.4. Using fractions we would write $\frac{27}{5} = 5 + \frac{2}{5}$, so we have $q = 5$ and $r = 2 = 27$ **mod** 5.

From mathematical to computer data types

We run into two problems when we try to make computer data types out of these mathematical ones:

- Mathematical sets, such as **Z** and **R**, are infinite and we cannot represent an infinite number of integers or real numbers with a finite amount of computer memory.
- Real numbers are stored in computer memory in a binary form. Many real numbers and fractions, such as $1/3$, π, or $\sqrt{2}$, have infinite decimal and binary representations, so they cannot be stored exactly in binary form with a finite amount of computer memory. Also, some numbers such as $1/10 = 0.1$ have finite decimal representations but infinite binary representations.

The solution to the first problem is to restrict the ranges of the numbers to a finite subset of integers or real numbers, and for the second problem we must simply accept the fact that there is some small **round-off error** (loss of precision) incurred when the infinite decimal expansions of certain numbers such as π or $\sqrt{2}$ are truncated to fit in a finite computer memory.

For storage, it is necessary to use a specific number of bits for each number. It is conventional to specify several kinds of numerical data types with different “sizes” (number of bits). The larger the number of bits, the larger the range of numerical data that can be represented. For integers, two common choices are 16-bit and 32-bit integers.

Similarly, real numbers are called **floating point numbers** and are commonly stored using 32 or 64 bits. The 32-bit numbers are called **single precision** floating point numbers. The 64-bit numbers are called **double precision** floating point numbers, since they can store numbers with approximately twice the precision (number of significant digits). As a rule of thumb, 32-bit floating point numbers have about 7 or 8 decimal digits of precision, and 64-bit numbers have about 16 decimal digits of precision. For example, in Java π has the approximate value 3.1415927 as a single precision number and 3.141592653589793 as a double precision number. The trade off is more precision at the expense of more storage space. Of course loss of precision (round-off error) does not occur for integer values; either the number fits exactly in memory or there is overflow.

In Java there are seven numeric data types. Five are integer data types, called `byte`, `short`, `char`, `int`, and `long`. The other two numeric types are floating point types, called `float` and `double`. These numeric types and the non-numeric boolean type are called **primitive types** since all data types are built from them. The seven numeric types, their sizes in bits, and their ranges, are shown in Table 2.1. From the table, the minimum and maximum values of the five integer types are -2^{n-1} and $2^{n-1} - 1$, where n is the number of bits. We normally use the standard `int` and `double` types to represent integers and floating point numbers.

Since our integers now have a limited range we sometimes need to be concerned with **overflow** after performing an arithmetic operation. For example, what happens if we add something to the largest `int` value? What happens if we add two `int` values and the sum is larger than the largest

Type	bits	Minimum value	Maximum value
`byte`	8	-128	127
`short`	16	-32768	32767
`char`	16	0	65535
`int`	32	-2147483648	2147483647
`long`	64	-9223372036854775808	9223372036854775807
`float`	32	$\approx \pm 1.40 \times 10^{-45}$	$\approx \pm 3.40 \times 10^{38}$
`double`	64	$\approx \pm 4.94 \times 10^{-324}$	$\approx \pm 1.80 \times 10^{308}$

Table 2.1: The Java integer and floating point primitive types.

`int` value? These problems with integer operations are called overflow. In other words, performing arithmetic operations on `int` values can cause overflow and the result will be undefined.

A similar problem occurs for floating point numbers of type `float` or `double`. For example, the numbers 3.4×10^{45} and -5.65×10^{56} have exponents that are outside the range of the `float` type but within the range of the `double` type. For floating point numbers there is also the problem of **underflow**. There are floating point numbers such as 1.5×10^{-400} that are not zero, but according to the table they are smaller than the smallest possible non-zero number. They are normally stored as zero. This is called underflow to 0.

It is not necessary to understand in detail how numbers are actually stored internally in binary form. It is only necessary to understand how overflow and round-off errors can occur.

The char data type

The `char` data type represents characters, such as the letters of the alphabet, punctuation, or digits. In Java a **character literal** is represented as a character enclosed in single quotes. For example, the letter “a” is denoted by `'a'`, the digit “3” is denoted by `'3'`, and the exclamation mark “!” is denoted by `'!'`. It may seen strange that the `char` type is considered to be a numeric type, since we don’t normally think of doing arithmetic on characters. However, internally, characters are represented as numbers. For example, the character `'A'` is represented by the 8-bit ASCII code 65, or in Unicode, which Java uses, by the 16-bit integer with value 65. We will not use the `char` data type in this Chapter.

The boolean data type

The only non-numeric primitive type is the `boolean` type which represents the two logical values for true and false, denoted in Java by the **boolean literals** `true` and `false`. We will not use this type until we discuss conditional statements in a later Chapter.

2.1.2 Integer and floating point literals

An integer value such as `0`, `4`, or `5434` is called an **integer literal**. Similarly, a floating point value with a decimal point such as `3.1416` is called a **floating point literal**. Scientific notation can also

Figure 2.1: Pictorial representation of uninitialized variables

be used for a floating point literal. For example, `6.023E23` denotes 6.023×10^{23} and uses an `E` to indicate the power of 10. Lowercase `e` can also be used to denote the exponent. Integer literals never have a decimal point and floating point literals always have one, except a decimal point is not required if an exponent is used. For example, `3E-5` is a valid floating point literal representing $3 \times 10^{-5} = 0.00003$.

2.1.3 Declaring and initializing variables in Java

The concept of a **variable** is fundamental to all programming languages. In Java, each variable has a name, a type (such as `int` or `double`), and it corresponds to a storage location in computer memory which can hold a value of the specified type. Think of a variable as a "named storage location". The content of this storage location is the value of the variable. Thus, at any stage in the execution of a computer program, a variable has a name and a value. To **declare** a variable means to specify its type and its name in a **declaration statement**. This provides information to the compiler that is used to allocate enough storage space.

■ EXAMPLE 2.1 **(variable declarations)** The statements

```
int width;
double area;
```

declare an `int` variable called `width` and a `double` variable called `area`. ■

The declarations in Example 2.1 allocate two storage locations: 32 bits for an integer value and 64 bits for a double precision value. They do not specify or define the content of the storage locations. The content of the storage location for a variable is called the **value** of the variable. In Example 2.1 `width` and `area` are **uninitialized variables**. We say that the value of an uninitialized variable is undefined. It is useful to have a pictorial representation of a variable as a "named storage location" in memory. This is illustrated in Figure 2.1 for the two uninitialized variables. The box represents the storage location; the variable name is written beside the box. The content of the box is the value of the variable. A question mark is shown to indicate that the two variables have not been initialized.

Before we can use these variables they need to be given values. There are two ways to do this: at the same time they are declared, or later in an assignment statement. We can give them values when we declare them as the following example shows.

■ EXAMPLE 2.2 **(variable declaration and initialization)** Instead of the declarations in the preceding example, we can use the **initialized declarations**:

```
int width = 5;
double area = 3.1416;
```

width
5
area
3.1416

Figure 2.2: Pictorial representation of initialized variables

Each declares a variable and gives it an initial value at the same time. The pictorial representation of these two variables, after initialization, is shown in Figure 2.2. The values in the boxes are the current values of the variables. ■

■ EXAMPLE 2.3 It is also possible to declare and initialize multiple variables of the same type in a single declaration:

```
double radius, area, circumference;
```

Another possibility is

```
double radius = 2.0, area, circumference;
```

which declares three variables and assigns the initial value `2.0` to `radius`. ■

Alternatively, when the variables are declared without initial values, we can initialize them later, as shown in the next example.

■ EXAMPLE 2.4 **(assignment statements)** If the variables `width` and `area` have been previously declared, as in Example 2.1, we can give them values using statements like:

```
width = 5;
area = 3.1416;
```

Each of these statements is an example of an **assignment statement**. ■

■ EXAMPLE 2.5 **(Multiple assignment statement)** It is possible to give several variables of the same type a common value using a multiple assignment statement. Assuming that `a`, `b`, and `c` have been declared as variables of type `double`, the statement

```
a = b = c = 0.0;
```

gives each the value `0.0` and is equivalent to

```
a = 0.0;
b = 0.0;
c = 0.0;
```

which uses three separate assignment statements. ■

Assignment statements always have the name of a variable to the left of the `=` sign and the value to be given the variable on the right of the equal sign. You can read the statement "`width = 5;`" as "`width` gets or receives the value 5". It is an error to use a variable on the left side of an assignment statement if it has not been previously declared in a declaration statement. You can always distinguish a declaration statement from an assignment statement because the former specifies the type and the latter does not specify it.

A characteristic feature of a variable is that it has a type and a value. Java is a **strongly typed** language. This means that once a variable has been declared, its type can never be changed. However, when the program is running the value of a variable can be changed at any time using an assignment statement.

If you forget to give a value to a variable and later attempt to use its value, the Java compiler will complain with an error message saying that the variable has not been initialized.

Rules for naming variables

In Java, the simplest kind of name is called an **identifier**. An identifier must begin with a letter, the dollar sign character (`$`), or the underscore character (`_`). Any remaining characters can also be one of these, or any digit character (0 to 9). It is recommended that you not use the dollar sign or underscore in variable names. Some identifiers, such as the type names `int` and `double`, are reserved words called **keywords**. They cannot be used for other purposes, such as variable names.

Java is also **case-sensitive**. This means that the case (uppercase or lowercase) is significant. For example, `width`, `Width`, and `WIDTH`, would be the names of three different variables. It is conventional to begin variable names with a lowercase letter and you should always follow this convention, even though it is not a language requirement.

Constants

In addition to variables, Java also has **constants**. Like variables they have names, a type, and a value. However once a value has been assigned it can never be changed. Constant declarations are distinguished from variable declarations by using the strange keyword `final`: once you have specified a value its final! The Java compiler will complain if you attempt to change the value of a constant. The main purpose of a constant is to give a meaningful name to a literal, such as an integer or floating point literal, as the following example shows:

■ EXAMPLE 2.6 **(declaring constants)** The declarations

```
final double CM_PER_INCH = 2.54;
final int MARGIN_WIDTH = 5;
```

define a `double` constant for a conversion factor from centimeters to inches, and an `int` constant that might represent the default width of the margin in a typesetting program. ■

Effective use of constants improves readability and makes it easier to modify programs. For example, if the number `5` appears in several places, it might mean the margin width in one place and something else in another place. Then, changing the margin width is difficult. It is easy if a constant is used.

It is also conventional to use all uppercase letters for constants with the underscore character simulating a space.

2.2 Arithmetic operations and expressions

2.2.1 Basic arithmetic operations

In Java the addition and subtraction operations are denoted as in mathematics by `+` and `-`. Juxtaposition cannot be used for multiplication since we want to have variable names longer than one letter: `ab` is a variable, not the product of `a` and `b`. Therefore, most programming languages use the asterisk `*` to denote multiplication. For example, `ab` is a variable but `a*b` is the product of variables `a` and `b`. Division is denoted by `/`. As in mathematics there are two kinds of division: integer division to obtain the quotient, and the real, or floating point, division. In pseudo-code **div** is often used for integer division and `/` is used for floating point division.

Unfortunately in Java `/` is used for both kinds of division: `a/b` is an integer division only if both `a` and `b` have integer values. If either or both of `a` and `b` have floating point values it is a floating point division. In Java, the modulus operation a **mod** b, giving the remainder when a is divided by b, is denoted by `a % b`. The `%` operator is called the modulus operator or the remainder operator.

2.2.2 Arithmetic expressions and precedence rules

Mathematical expressions involving the basic operations can easily be translated to Java using the same well known mathematical **precedence rules** (or order of operations):

1. `*`, `%`, and `/` have the same precedence. They have a higher precedence than `+` and `-` so they are done first, in the left to right order in which they appear. This is called **left associativity**. Example: In the expression `a + b*c + d` the multiplication is done first.

2. `+` and `-` have the same precedence and are done next in the left to right order in which they appear. They are also left associative. Example: In the expression `a + b - c`, the addition is done first, followed by the subtraction.

3. Parentheses have the highest precedence of all and can change the precedence of the other operators. For example, in the expression `a + b*c + d` the multiplication is done first, but in the expression `(a + b)*(c + d)`, the multiplication is done last.

There are many other operators so this table is not complete.

■ EXAMPLE 2.7 **(unary and binary operators)** The arithmetic operators `+`, `-`, `*`, `/`, and `%` are binary operators. For example, in an expression such as `a + b`, the values `a` and `b` are said to be the **operands** of the `+` operator. Since there are two operands, `+` is called a **binary** operator. Unlike the binary `*` and `/` operators, the `+` and `-` operators can also be used in expressions such as `-b`, or `+b`, where they have only one operand, the value of `b`. In this context they are called **unary operators**.
■

■ $\boxed{\text{EXAMPLE 2.8}}$ **(mathematical expressions)** Table 2.2 shows some mathematical expressions and their translations to Java, assuming that all variables are real and represented as type `double`. You must be careful with division operations. If $\frac{9}{5}$ and $\frac{5}{9}$ had been translated as `9/5` and `5/9`, the wrong results would be obtained, since `9/5` is an integer division with value `1`, and `5/9` is an integer division with value `0`. ■

Mathematical Expression	**Java Expression**
$a+bc-4$	`a + b*c - 4.0`
$\frac{1}{2}(a+b)(c-7)$	`(a + b)*(c - 7.0)/2.0`
$\frac{9}{5}c+32$	`(9.0/5.0)*c + 32.0`
$\frac{5}{9}(f-32)$	`(5.0/9.0)*(f - 32.0)`
$a^2b^2+\frac{c^3}{a+b}$	`a*a*b*b + c*c*c/(a + b)`
$3x^2-2x+4$	`3.0*x*x - 2.0*x + 4.0`
$1.3+x(3.4-x(2.5+4.2x))$	`1.3 + x*(3.4 - x*(2.5 + 4.2*x))`
$s(s-a)(s-b)(s-c)$	`s*(s - a)*(s - b)*(s - c)`

Table 2.2: Translation of mathematical expressions to Java.

■ $\boxed{\text{EXAMPLE 2.9}}$ **(integer vs floating point division)** Let `totalCents` be a variable of type `int` having the value `3527`. Then `totalCents / 100` is an integer division having the value `35` and `totalCents % 100` gives the remainder 27. On the other hand, `totalCents / 100.0` is interpreted as a double floating point result, so its value is `35.27` since 100.0 is a double literal constant.
■

In these examples of expressions, as a matter of style, we have surrounded the binary operators `+` and `-` by a single space, but we have not done this for the binary operators `*` and `/`. This is a common convention to emphasize that a typical arithmetic expression is composed of terms and factors. The terms are separated by the binary `+` and `-` operators, and the factors are separated by `*`, `/` and `%`. In the first example in Table 2.2 the terms are `a`, `b*c` , and `4.0`. The term `b*c` is composed of two factors `b` and `c`. The extra space around terms makes them stand out. If you prefer, you can surround all binary operators by a single space.

2.3 Assignment statements

Assignment statements are used to give values to variables whose type has already been declared. In Java the `=` sign is used to indicate an assignment. The left side is the name of the variable and the right side is an expression. In the simplest cases an expression may be a literal, or a variable, or an expression involving arithmetic operators. When an assignment statement is executed, the value of the expression on the right side is evaluated and assigned as the value of the variable on the left side.

■ EXAMPLE 2.10 **(= does not denote mathematical equality)** An assignment statement should never be confused with an equation or an equality. There is a big difference between the assignment statement

```
x = x + 1;
```

and the mathematical equation $x = x + 1$. In the assignment statement the right side is evaluated by adding one to the current value of `x` and assigning the result as the new value of `x`. The mathematical equation is meaningless since it implies that $0 = 1$. ■

■ EXAMPLE 2.11 **(special combination operators)** There are also special combination operators such as `+=`, `-=`, `*=`, and `/=`, which combine an arithmetic operation and assignment. For example, the assignment statement

```
totalArea += area;
```

is just shorthand for

```
totalArea = totalArea + area;
```

We will try to avoid using these combination operators since they make programs harder to read. ■

■ EXAMPLE 2.12 **(assignment statements)** The following statements declare three variables of type `double`, using one combined declaration instead of three, and use three assignment statements to specify the radius and calculate the area and circumference of a circle having this radius:

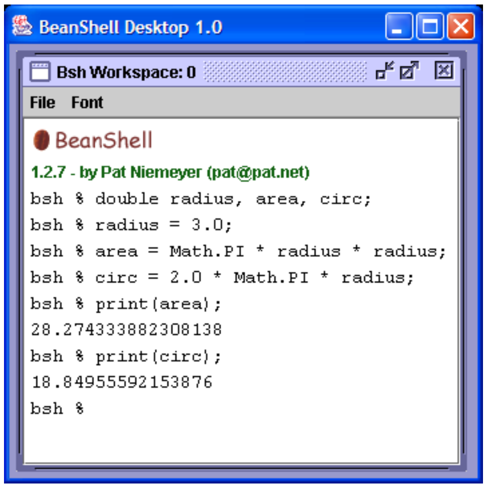
```
double radius, area, circ;
radius = 3.0;
area = Math.PI * radius * radius;
circ = 2.0 * Math.PI * radius;
```

It is not necessary to remember the double precision value of π, since it is available as the constant `Math.PI` in a special built-in Java class called `Math`. We will see that there are many useful mathematics functions in the `Math` class. The name `Math.PI` is an example of a qualified name (a name with a dot in it). ■

2.3.1 Try it with BeanShell

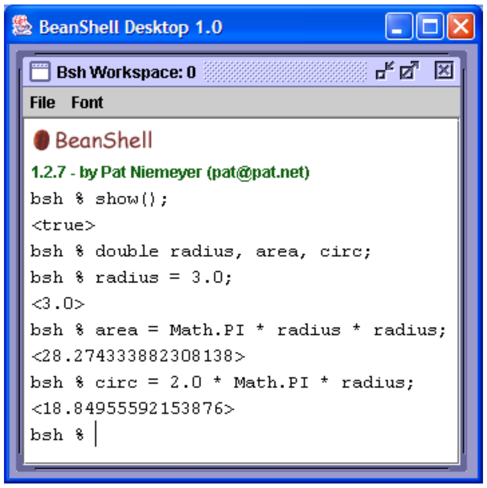
You can test your understanding of simple examples like this using the interactive BeanShell program: a scripting shell for Java. With it you can execute Java statements immediately.

In Figure 2.3 we show BeanShell in action on Example 2.12. In Figure 2.3(a) BeanShell's special `print` statement is used to see the value of a variable or expression. In Figure 2.3(b) BeanShell's special `show()` function is used to automatically show the result of each assignment statement in angle brackets. This function acts like a "toggle" that turns on or off the displaying of intermediate results from assignment statements. This is the most useful way to get output for simple examples. If show is "on" then you can see the value of a variable at any time by simply typing its name followed by a semicolon.



```
bsh % double radius, area, circ;
bsh % radius = 3.0;
bsh % area = Math.PI * radius * radius;
bsh % circ = 2.0 * Math.PI * radius;
bsh % print(area);
28.274333882308138
bsh % print(circ);
18.84955592153876
bsh %
```

(a)



```
bsh % show();
<true>
bsh % double radius, area, circ;
bsh % radius = 3.0;
<3.0>
bsh % area = Math.PI * radius * radius;
<28.274333882308138>
bsh % circ = 2.0 * Math.PI * radius;
<18.84955592153876>
bsh %
```

(b)

Figure 2.3: Using BeanShell to display results.

The `print` and `show` functions are not part of Java. They are simply provided by BeanShell to control and view output. Later we will see how to do output in Java.

■ EXAMPLE 2.13 **(dollars and cents)** Continuing with Example 2.9 and assuming that `show` is 'on'. try the following statements in BeanShell

```
bsh % int totalCents, cents, dollars;
bsh % totalCents = 3527;
<3527>
bsh % dollars = totalCents / 100;
<35>
bsh % cents = totalCents % 100;
<27>
```

Here `bsh %` is the BeanShell prompt and is not typed ■

■ EXAMPLE 2.14 **(integer division and remainder)** Try the following assignment statements in BeanShell, assuming that `show()` is in 'on'.

```
bsh % int n = 123, remainder, hundreds, tens, units;
bsh % hundreds = n / 100;
<1>
bsh % remainder = n % 100;
<23>
bsh % tens = remainder / 10;
<2>
bsh % units = remainder % 10;
<3>
```

Here a multiple declaration statement is used to declare five variables and initialize one of them. Then the hundreds, tens, and units digits are extracted from the 3-digit integer `123`. ■

■ EXAMPLE 2.15 **(special increment and decrement operators)** In Java there is a special increment operator, denoted by `++`, for adding one to the value of a variable, and a special decrement operator, denoted by `--`, for subtracting one from the value of a variable. Try the following statements in BeanShell, assuming `show()` is 'on'.

```
bsh % int i = 3;
bsh % int j = 4;
bsh % i++;
<3>
bsh % print(i);
4
bsh % j--;
<4>
bsh % print(j);
3
bsh %
```

Notice that the automatically displayed values are the values before the increment and decrement is applied. The BeanShell `print` statement shows the final values. The statements

```
i++;
j--;
```

are equivalent to the assignment statements

```
i = i + 1;
j = j - 1;
```

We will not use the increment and decrement operators too often, since they can make programs harder to read. There are also `--j` and `++j` forms of these operators. Also in some cases `++j` and `j++` have different effects and similarly for `--j` and `j--`. ■

2.4 Conversion between numeric types (type casting)

It is often necessary to convert a value of one numeric type to a value of another type. Sometimes the compiler will automatically do this and sometimes it will complain and produce an error message. The basic rule can be obtained from Table 2.1 in the column that indicates how many bits of storage are required for the values of each type. If you attempt to convert to a type which requires a smaller number of storage bits, the compiler will issue an error message because information may be lost in converting to a value with a smaller number of bits. However we can force the conversion with a technique called **type casting**, illustrated in the examples below. Of course this is normally useful only if information is not lost in converting to the smaller size so numeric type casts should be used with care.

■ EXAMPLE 2.16 **(valid implicit type conversions)** Assume that `d` and `e` have type `double` and `i` has type `int`. The two assignment statements in

```
bsh % int i = 1;
bsh % double d, e;
bsh % d = i;
<1>
bsh % e = i + 3.55;
<4.55>
```

do not cause any problems. In the first assignment the integer `i` is being converted to a larger size (any `int` will fit in a `double`). In the second assignment, to add the 32-bit integer value of `i` to the 64-bit double value `3.55`, the value with the smaller size is first converted to a value of the larger size (so the value of `i` is converted to a double), with no loss of information, then the addition is performed as a 64-bit addition and the result is assigned to `e`. ■

■ EXAMPLE 2.17 **(invalid implicit type conversions)** Continuing the previous example the assignment statements in

```
bsh % int j;
bsh % i = e;
// Error: Typed variable: i: Can't assign double to int: ...
bsh % j = i + e;
// Error: Typed variable: j: Can't assign double to int: ...
```

each result in an error message. The first statement is an attempt to assign a 64-bit number as the value of a 32-bit variable. This cannot always be done without losing information. The second statement is an attempt to add the 32-bit integer `i` and the 64-bit double number `d`, and assign the result as the value of a 32-bit integer `j`. The expression on the right side does not cause problems: the 32-bit value of `i` can be converted to a 64-bit double precision number and then added to the value of `d`. However, the attempt to assign this double precision result to `j` may cause a loss of information so the result is a compiler error. ■

2.4.1 Truncation of floating point numbers

A type cast can be used to truncate a floating point number. This has the affect of discarding the fractional part of the number to produce an integer result. Even though there is a loss of information, this is often a useful operation.

■ EXAMPLE 2.18 **(explicit type conversion as truncation)** Continuing with the previous example try

```
bsh % i = (int) e;
<4>
bsh % j = i + (int) e;
<8>
```

and the compiler does not produce any errors. The use of an explicit type name in parentheses in front of an expression is called a **type cast**. It tells the compiler to do the conversion anyway even if data could be lost. A type cast is a unary operator that converts the expression to which it is applied, to the specified type. Therefore, "`(int) e`" causes the double number `e` to be converted to an `int` type. This means that the fractional part of `e`, if any, is thrown away and the resulting integer is kept as the value. This is called **truncation** and is a useful operation if we want the integer part of a real number. If the integer is too big to store in an `int` then garbage is produced. For example, in BeanShell try

```
bsh % i = (int)12345.5434;
<12345>
bsh % i = (int) 12345678912343.5;
<2147483647>
```

and observe that the value `12345.5434` is truncated to `12345`, which is not too large for a 32-bit integer. However, the `double` value `12345678912343.5` would be truncated to `12345678912343` which is too large to hold in a 32-bit integer and overflow occurs with a meaningless result. In fact the result here is the largest integer value (see Table 2.1.) ■

2.4.2 Loss of precision in floating point conversions

A loss of precision can also occur when trying to convert values of type `double` to type `float` since the size is reduced from 64 bits to 32 bits.

■ EXAMPLE 2.19 **(explicit type conversion as loss of precision)** If `d` has type `double` and `f` has type float then the statement

```
f = d;
```

results in a compiler error. We can use the typecast

```
f = (float) d;
```

The result will generally be a loss of precision in the conversion (see Table 2.1) which may be acceptable in some applications. Consider the BeanShell example

```
bsh % double d = 1.11111111111111;
bsh % float f;
bsh % f = d;
// Error: Typed variable: f: Can't assign double to float: ...
bsh % f = (float) d;
<1.1111112>
bsh % d = 1e-66;
<1.0E-66>
bsh % f = (float) d;
<0.0>
bsh % d = 1e66;
```

```
<1.0E66>
bsh % f = (float) d;
<Infinity>
```

The first typecast results in a loss of precision, giving `1.1111112`, which may be acceptable if we don't require the full precision of the `double` type. The second typecast gives an exponent underflow so the float value will be `0`, and the last typecast gives an exponent overflow so the the float value will be infinity. In the last two cases the exponents are outside the range of the `float` type (see Table 2.1). ■

2.5 Arithmetic functions from the `Math` class

We will see that Java programs are made up of one or more classes each of which contains methods (functions are called methods in Java). Java has many built-in libraries of useful classes and methods. For example, the `Math` library contains many standard mathematical functions as well as two constants. We have already used the constant `Math.PI` to represent the double precision value of π. There is also `Math.E` which represents the double precision value of e.

There is a function `Math.sqrt(x)` for computing $\sqrt{x}$; a power function `Math.pow(x,y)` for computing x^y; the trigonometric functions `Math.sin(x)`, `Math.cos(x)`, and `Math.tan(x)`; the inverse trigonometric functions `Math.asin(x)`, `Math.acos(x)`, and `Math.atan(x)`; the exponential and log functions `Math.exp(x)` and `Math.log(x)`, and several other functions.

2.5.1 Examples of athematical functions

Here are some examples that use mathematical functions. BeanShell can be used to try them.

■ EXAMPLE 2.20 **(square root function)** The statements (assuming show is 'on')

```
bsh % double x = 1.0, y = 2.0;
bsh % double x1 = 1.0, y1 = 2.0, x2 = 2.0, y2 = 3.0;
bsh % double distance1 = Math.sqrt(x*x + y*y);
bsh % double dx = x2 - x1;
bsh % double dy = y2 - y1;
bsh % double distance2 = Math.sqrt(dx*dx + dy*dy);
bsh % print(distance1);
2.23606797749979
bsh % print (distance2);
1.4142135623730951
```

compute the distance $\sqrt{x^2+y^2}$ of the point (x,y) from the origin, and the distance $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$ between the points (x_1,y_1) and (x_2,y_2) using the specific points $(x,y) = (1,2)$, $(x_1,y_1) = (1,2)$, and $(x_2,y_2) = (2,3)$. ■

■ EXAMPLE 2.21 **(distances using the square root and cosine functions)** Given that `a` and `b`, are two sides of a triangle, and `gamma` is the contained angle in degrees, the declaration statements

```
bsh % double a = 1.0, b = 1.0, gamma = 90.0;
bsh % gamma = Math.toRadians(gamma);
<1.5707963267948966>
bsh % double c = Math.sqrt(a*a + b*b - 2.0*a*b*Math.cos(gamma));
bsh % double perimeter = a + b + c;
bsh % double s = perimeter / 2.0;
bsh % double area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
bsh % print(perimeter);
3.414213562373095
bsh % print(area);
0.5
bsh % print(c);
1.414213562373095
```

compute the length of the third side and the perimeter and area of the triangle with side lengths `1` and contained angle `90` degrees. Note that the trigonometric functions sin, cos, and tan require angles in radians so we have use the function `Math.toRadians` to do the conversion. There is also a function `Math.toDegrees` to convert radians to degrees. ■

BeanShell tip: If you are typing more than a few statements into the BeanShell workspace, as in the previous two examples, and make a mistake then you may have to start over. To avoid this open a workspace editor from the "File menu". A mini-editor appears and you can type your statements here and edit them. When you want to execute the statements just select "Eval in Workspace" from the "Evaluate" menu and the results will appear in the workspace.

■ EXAMPLE 2.22 **(calculating windchill)** Given the wind speed `v` in kilometers per hour and the air temperature `t` in degrees Celsius, the statement

```
double wc = 0.045*(5.27*Math.sqrt(v) + 10.45 - 0.28*v)*(t - 33.0) + 33.0;
```

computes the wind chill temperature in kilometers per hour. If you are using miles per hour for `v` and degrees Fahrenheit for `t` then

```
double wc = 0.0817*(3.71*Math.sqrt(v) + 5.81 - 0.25*v)*(t - 91.4) + 91.4;
```

would be the appropriate statement. ■

■ EXAMPLE 2.23 **(a heat loss formula)** Another formula that computes heat loss instead of windchill is given by

```
double h = (10.45 + 10.0*Math.sqrt(v) - v)*(33.0 - t);
```

where `v` is the wind speed in meters per second, `t` is the temperature in degrees Celsius, and `h` is the heat loss in kilo calories per square meter per hour. ■

■ EXAMPLE 2.24 **(the power function)** The expression $x^{2/3} + y^{2/3}$ can be computed with the declaration statement

```
double d = Math.pow(x, 2.0/3.0) + Math.pow(y, 2.0/3.0);
```

assuming that the double variables `x` and `y` have been assigned values. ■

■ $\boxed{\text{EXAMPLE 2.25}}$ **(investment example using the power function)** If r is the interest rate in percent per year, m is the number of times interest is compounded per year, a is the initial investment, and n is the number of years, then the Java translation of the value

$$v = a\left(1 + \frac{r}{100m}\right)^{mn}$$

of the investment after n years is

```
double v = a * Math.pow(1.0 + r / (100.0*m), m*n);
```

using the `Math.pow` function in the `Math` class. The inner parentheses are very important here. Without them it would be a division by `100.0` followed by a multiplication by `m`, and not a division by `100.0*m` as required. Also both arguments of the `pow` function are of type `double`. We specified `m*n`, which is an `int`, as second argument and the compiler does an implicit type cast to convert it to a `double` argument value. ■

■ $\boxed{\text{EXAMPLE 2.26}}$ **(using exp, sin, and cos)** You may have seen expressions such as $e^{-3x}\cos x - 2e^{-2x}\sin x$ in calculus. The Java translation is

```
Math.exp(-3.0*x)*Math.cos(x) - 2.0*Math.exp(-2.0*x)*Math.sin(x)
```

assuming that the `double` variable `x` has been assigned a value. ■

■ $\boxed{\text{EXAMPLE 2.27}}$ **(generating random integers)** The `Math.random()` method returns a random double precision number r such that $0 \le r < 1$. For example the following statement

```
int number = (int) (10 * Math.random()) + 1;
```

generates a random integer in the range 1 to 10 and assigns it as the value of `number`.

First the integer `10` is converted to a `double` value, and the double precision multiplication is performed to give a value r in the range $0 \le r < 10$. Then the type cast converts this by truncation to an integer i, in the range $0 \le i \le 9$, and 1 is added to give an integer in the range in the range $1 \le i \le 10$. which is assigned to `number`. The parentheses around `10 * Math.random()` are necessary since the type cast `(int)` is always applied to the value on its immediate right, which would be `10` without the parentheses. ■

2.5.2 Rounding floating point numbers

We have seen how to truncate floating point numbers to obtain integers using a type cast in Example 2.18 and Example 2.27. Sometimes it is necessary to round floating point numbers to the nearest integer. The `Math.round` function will round a `double` number to the nearest integer. The return type is `long` not `int` since rounding a `double` may produce a value that will not fit in a 32-bit integer but will always fit in a 64-bit `long` integer.

■ EXAMPLE 2.28 **(rounding to an integer)** Consider the following statements in BeanShell:

```
bsh % int i, j, k;
bsh % i = (int) Math.round(123.45);
<123>
bsh % j = (int) Math.round(123.56);
<124>
bsh % k = (int) Math.round(-123.56);
<-124>
bsh % k = Math.round(123.56);
// Error: Typed variable: k: Can't assign long to int: ...
```

The rounded value of `123.45` is `123` as a 64-bit integer, the rounded value of `123.56` is `124` as a 64-bit integer, and the rounded value of `-123.56` is `-124` as a 64-bit integer. The final assignment statement shows that it is necessary to use the typecast `(int)` to force conversion from `long` to `int`. ■

■ EXAMPLE 2.29 **(rounding to two decimal places)** A double precision variable `x` can be rounded to two digits after the decimal point using a declaration statement such as:

```
double x2 = Math.round(x * 100.0) / 100.0;
```

If `x` has the value `123.4567` then `x * 100.0` has the value `12345.67`, and `Math.round` converts this to the `long` integer `12346`, and division by `100.0` produces the `double` number `123.46`. ■

2.5.3 Mathematical function prototypes

In Java functions are called methods. To understand how to use one of the mathematical functions in the `Math` class, we need to know (1) the name of the method, (2) the formal arguments and their types, if any, and (3) the type of value that is computed and returned by the function.

For example, the function `Math.pow` computes x^y, so it needs two arguments which we can call `x` and `y`, they are both `double` numbers, and the value returned has type `double`. This tells us immediately that the assignment statement in Example 2.25 has the correct form (syntax), recalling that the compiler will implicitly convert the `int` argument `m*n` to a `double` value.

In Java the **method prototype** is used to give a compact description of the rules for using the method. For the power function the prototype is:

```
static double pow(double x, double y)
```

The word `static` means that this method is not associated with any object (see Chapter 3). Next we have the return type (`double`), and then the method name (`pow`). Inside the parentheses there is a list of argument types followed by some names (`x` and `y`) to refer to the arguments.

Each type-name pair such as `double x` or `double y`, which appear in the method prototype, is called a **formal argument** (the word parameter is often used as a synonym for argument).

Each value supplied when the method is called is referred to as an **actual argument**. The process of using a method prototype to determine how to use a method is illustrated in Figure 2.4, for the declaration statement in Example 2.25. Here the rightmost two arrows show the corre-

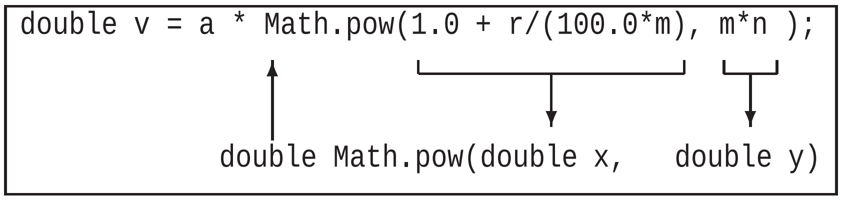

double v = a * Math.pow(1.0 + r/(100.0*m), m*n);
double Math.pow(double x,   double y)

Figure 2.4: Matching actual and formal arguments

spondence between the formal arguments and the actual arguments. The actual argument, `1.0 + r/(100.0*m)`, is an expression that corresponds to the formal argument `x`, and the actual argument, `m*n`, is an expression that corresponds to the formal argument `y`. The leftmost arrow shows that the value of the expression

```
Math.pow(1.0 + r/(100.0*m), m*n)
```

is a `double` number, so it makes sense to multiply it by the `double` number `a`.

The important idea here is that we can look at a statement, such as the one in Example 2.25, and immediately see, by looking at the prototype, that it is a valid use of the method. Each of the methods in the `Math` class has a prototype.

■ EXAMPLE 2.30 **(some math function prototypes)** Here are prototypes for some of the arithmetic functions we have used in the preceding examples:

```
static double sqrt(double x)
static double pow(double x, double y)
static double sin(double x)
static double cos(double x)
static double tan(double x)
static double exp(double x)
static double random()
static long round(double x)
```

Most of these functions take a single formal argument of type `double` and return a `double` value, except for `pow`, `random` and `round`. The `random` function takes no arguments, but the empty set of parentheses is still needed when calling the function (See Example 2.27).

The prototype for `round` clearly shows, as mentioned above, that the return value is of type `long`, not `int`, so that no information is lost in the rounding. ■

2.6 Terminology introduced in this chapter

In this section we give simple definitions of the most important concepts introduced in the Chapter.

simple identifier

A sequence of one or more letters, digits and underscores such that the first character is not a letter. Identifiers are used to give names to variables, classes and other entities.

An almost universal convention is to begin the name of a class with an upper case letter. All other identifiers begin with a lower case letter. In either case capitalize the beginning letter of each interior word.

Identifiers are **case sensitive**. The only example of a class we have seen so far is the `Math` class.

Examples: `radius`, `numberOfStudents`, `Math`

numeric literal

A value representing a number such as an integer or a floating point constant in fixed or scientific form

Examples: `1`, `-34` are literals of type `int`.
Examples: `1L`, `-3456789212231L` are literals of type `long`.
Examples: `1.0`, `-3.4`, `-3.4D`, `4.5`, `4.5D`, `4.5d`, `1.23E-04` are double literals. The suffix `d` or `D` is optional. An exponent (power of 10) is denoted by `e` or `E`.
Examples: `1.0f`, `-3.4f`, `-3.4F` are literals of type `float`. The suffix `f` or `F` must be present.

variable

A named storage location that can hold a value of some type.

Examples: `radius`

type

A specific kind of data such as the set of all integers or the set of all real numbers.

Examples: `int`, `float`, `double`

variable declaration

A statement having one of the forms

typeName identifier`;`
typeName identifier `=` *expression*`;`

where *identifier* is the name of the variable, *typeName* is the variable type and *expression* is an expression that evaluates to a value that can be assigned to the variable.

Example: `double radius;`
Example: `double radius = 2.0;`
Example: `double area = Math.PI * radius * radius;`
Example: `int n=123, remainder, hundreds, tens, units;`
Example: `double area, circumference;`
Example: `double radius = 3.0, area;`

The final three examples show that multiple variables of the same type can be declared and optionally initialized in a single declaration.

constant declaration

A constant has the form

`static final` *typeName identifier* `=` *expression*`;`

The strange keyword `static` indicates that constants are associated with the class, not the objects of the class. The equally strange keyword `final` distinguishes a constant declaration from an initialized variable declaration.

It is conventional to name constants using upper case letters and the underscore to simulate a space.

Example: `final double CM_PER_INCH = 2.54;`

arithmetic expression

An expression involving variables and operators that evaluates to a numeric value.

Example: `radius`
Example: `2.0 * Math.PI * radius`
Example: `remainder % 10`

assignment statement

A statement of the form

identifier `=` *expression* `;`

where *identifier* is the name of a variable that has already been declared and *expression* is an expression whose value is assigned to the variable.

Example: `radius = 2.0;`
Example: `area = Math.PI * radius * radius;`
Example: `a = b = c = 0.0;`

The last example shows that several variables can be assigned the same value in a multiple assignment statement.

2.7 Review exercises

▶ **Review Exercise 2.1** Define the following terms and give examples of each.

data type	Q-R theorem	`div`
`mod`	primitive types	floating point number
single precision	double precision	overflow
round-off error	underflow	truncation operation
scientific format	fixed point format	variable
rounding operation	variable	variable declaration
uninitialized variable	initialized declaration	assignment statement
identifier	keyword	case sensitive
precedence rules	increment operator	decrement operator
type casting	implicit type conversion	explicit type conversion
strongly typed	numeric literal	type
constant declaration	arithmetic expression	assignment statement
character literal	boolean literal	unary operator
binary operator	integer division	floating point division
`Math` class	method prototype	formal argument
actual argument		

▶ **Review Exercise 2.2** Express the following numbers as Java literals of type `float`:

(a) 1234567, (b) 1.9×10^{-37}, (c) 0.000045659043, (d) 3.14159

▶ **Review Exercise 2.3** Express the following numbers as Java literals in scientific format of type `double`:

(a) 1234567, (b) 1.9×10^{-37}, (c) 0.000045659043, (d) 3.14159

▶ **Review Exercise 2.4** Translate the following mathematical expressions or formulas into Java

(a) $\sqrt{x^{3/2}+y^{3/2}}$ (b) $x^2+y^2\left(\frac{1}{1+x^2}\right)^{1/2}$ (c) $\pi a\sqrt{a^2+h^2}$

(d) $3.2+4.7x+3.2x^2-7.5x^3$ (e) $e^{3x+2y}\sin(x+4y)$ (f) $\dfrac{x^2}{1+\sqrt{1+x^2}}$

(g) $\dfrac{\tan x+\tan y}{1-\tan x\tan y}$ (h) $m\left(\frac{v^2}{L}+g\cos\theta\right)$ (i) $c\left(\frac{x}{|x|}-\frac{x}{(x^2+y^2)^{1/2}}\right)$

▶ **Review Exercise 2.5** What are the differences among the three statements

```
int width;
int width = 5;
width = 5;
```

▶ **Review Exercise 2.6** Why would the compiler complain about the statements

```
int totalCents = 3527;
int dollars = totalCents / 100.0;
```

Write the correct statements.

► **Review Exercise 2.7** Give short answers to the following questions.

(a) What is the purpose of an assignment statement?

(b) What is the difference between the equal sign in an assignment statement and the equal sign that is often used to denote an equation?

(c) When does $x = x + 1$ make sense?

► **Review Exercise 2.8** Give short answers to the following questions.

(a) Why do round-off errors occur?

(b) What is overflow and how does it occur?

(c) What is underflow and how does it occur?

(d) What are the differences in Java among `0.1F`, `0.1`, and `0.1D`?

(e) If no suffix is specified what does the compiler assume about a number's type?

(f) In a mathematical expression, how would you change the order of precedence so that an addition operation is performed before a multiplication operation?

(g) In an expression in which an integer is multiplied by a `double` number, what type does the compiler assign to the result?

(h) Why does the compiler not automatically convert a `double` value to an `int` value?

(i) When truncating a value of type `double` by type casting to a value of type `int`, what error can occur?

(j) The prototype for the `sqrt` method indicates that the argument is of type `double`. Why does the method call expression `Math.sqrt(2)`, which uses an `int` argument not cause a compiler error?

(k) Give some examples of implicit type conversions.

(l) What is the difference between explicit type conversion and implicit type conversion?

► **Review Exercise 2.9** What is the largest number that can be correctly multiplied by itself before integer overflow occurs?

2.8 BeanShell exercises

► **BeanShell Exercise 2.1 (Evaluating mathematics formulas)**
For each expression in Review Exercise 2.4 pick some values for the variables and evaluate the expression.

► **BeanShell Exercise 2.2 (Converting inches to feet and inches)**
Write some statements that define a number of inches as an `int` variable `totalInches` and convert this number of inches to feet and inches and print the result. For example, 67 inches is 5 feet and 7 inches.

► **BeanShell Exercise 2.3 (Converting floating point hours to hours, minutes, and seconds)**
Write some statements that define a `double` variable `totalHours` initialized to some floating point number of hours such as `3.245`, convert it to hours, minutes and the nearest second, and display the results. For example, `3.245` hours is `3` hours, `14` minutes, and `42` seconds.

► **BeanShell Exercise 2.4 (Astronomy calculations)**
In astronomy, angles are measured in degrees, minutes (1/60 degree), and seconds (1/60 minute). Write some statements that define a given angle specified by three integers (degrees, minutes and seconds), convert it to a floating point angle, calculate the sine of this angle, and display the results. For example, for integer values `3`, `15`, and `45` the floating point angle is `3.2625` and the sine of this angle is `0.056910601485907715`.

► **BeanShell Exercise 2.5 (Celsius to Fahrenheit temperature conversion)**
Write some statements that define a temperature in degrees Celsius as a `double` value, convert it to degrees Fahrenheit, and display the converted temperature.

► **BeanShell Exercise 2.6 (Fahrenheit to Celsius temperature conversion)**
Write some statements that define a temperature in degrees Fahrenheit as a `double` value, convert it to degrees Celsius, and display the converted temperature.

► **BeanShell Exercise 2.7 (Pythagorean theorem)**
Write some statements that define the coordinates of two points (x_1, y_1) and (x_2, y_2) as `double` values, compute the distance between the points, and display the result.

► **BeanShell Exercise 2.8 (Height in metric units)**
Write some statements that define two `int` variables for the height of a person in feet and inches, convert the height to centimeters and display the result. For example, someone who is 5 feet 10 inches tall is 177.8 cm tall.

► **BeanShell Exercise 2.9 (Heat loss calculator)**
Write some statements that define a temperature in degrees Celsius and a wind speed in kilometers per hour as `double` variables and use the formulas in Example 2.22 and Example 2.23 to compute and display the wind chill temperature and the heat loss. To use the heat loss formula in Example 2.23 you will have to convert the input wind speed from kilometers per hour to meters per second.

For example, if the wind speed is 30 km/hr and the temperature is -15 C then the windchill is -33.77635416592807 and the heat loss is 1487.240646055102

Chapter 3

Writing Simple Classes

Using BeanShell and BlueJ

Outline

Writing simple Java classes using BlueJ

Experimenting with and testing classes using BlueJ

Using BeanShell with Objects and Methods

Understanding the structure of simple classes

Writing class documentation for Javadoc

Understanding common syntax and logical errors

Understand basic object-oriented terminology

3.1 Introduction

So far we have been writing simple sequences of statements that declare variables and assign the results of arithmetic expressions to them. We have used `BeanShell` to execute such sequences of statements.

In Java a program (application) consists of one or more classes that are used to construct objects which can interact with each another at execution time. Each class defines the functionality or behavior of its objects by defining a number or methods (functions) that an object can execute.

These abstract concepts can be quite confusing for the beginner. Fortunately, there is an integrated development environment (IDE) called `BlueJ` that lets us write classes and interactively construct objects and invoke methods on them. It is a leaning tool for understanding the three fundamental object oriented concepts of **class**, **object**, and **method**. In this sense `BlueJ` is unlike other development environments. Also, with `BlueJ` it is very easy to test our classes and when we are finished we can package our application in a form that can be executed outside the `BlueJ` environment.

3.2 `CircleCalculator` class using `BlueJ`

In this section we begin our study of object-oriented programming using `BlueJ` to write a simple class called `CircleCalculator`, using the formulas in Example 2.12 to define how to calculate the area and circumference of a circle.

In `BlueJ` we have the concept of a **project**. It is a directory (folder) that contains all the class files (java source files, byte code files) and other files associated with your project. You can create a project and type in the classes yourself or you can use the `BlueJ` projects supplied with this book that already contain them.

For this chapter we assume that the `CircleCalculator` class is found in a `BlueJ` project called `book-projects/chapter3`. The other classes in this Chapter, `TriangleCalculator` and `QuadraticRootFinder`, are also in this project.

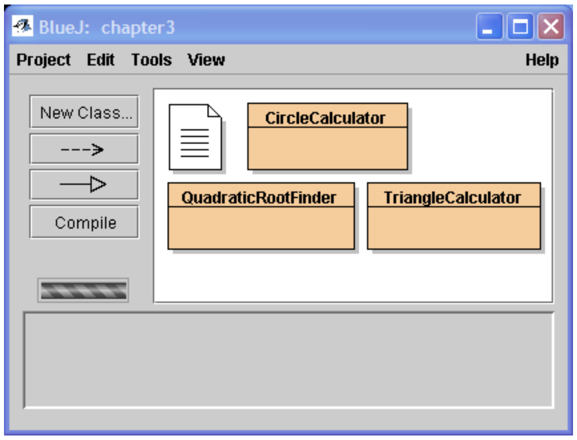
3.2.1 Experimenting with the class

Before analyzing the source code for this class we can experiment with the class by launching `BlueJ` and opening the `chapter3` project to get the display shown in Figure 3.1(a). Now perform the following steps to test the class:

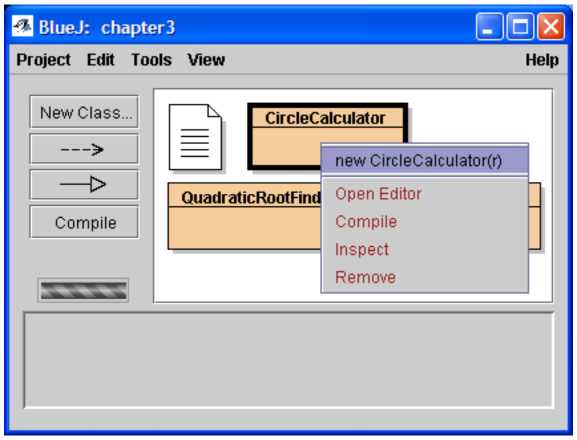
1. The `CircleCalculator` rectangle represents the class. Before using the class for the first time its source code file must be compiled into an object code (bytecode) file. You will know when compilation is necessary because the class rectangle will be diagonally shaded. If this is the case you can select the compile button to compile it, or right click on the class rectangle and select compile.

2. Now right click on the `CircleCalculator` rectangle to bring up its menu.

3. Select the `new CircleCalculator` menu choice shown in Figure 3.1(b).



(a)



(b)

Figure 3.1: (a) shows the `chapter3` project with `CircleCalculator` highlighted,(b) shows how to right-click to get the constructor menu.

4. In the "Create Object" dialog box shown in Figure 3.2(a) you can give a name to the object or accept the default name. We have used the name `circle1`. Also you must provide a value for the radius and we have chosen 2 When you select OK an object of the class is constructed. The new object will appear on the object workbench as a red rounded rectangle (see Figure 3.2(b)). The name of the object and the name of the class are shown in this object box. Each object created from the class is called an **instance** of the class.

5. Right click on this object and select a method from the menu shown in Figure 3.2(b). If you select `double getArea()` then the `getArea` method will be executed and the result for the area of the circle with radius 2 will be as shown in a message box (see Figure 3.3).

6. Now repeat the previous step and execute the `getCircumference` and `getRadius` methods.

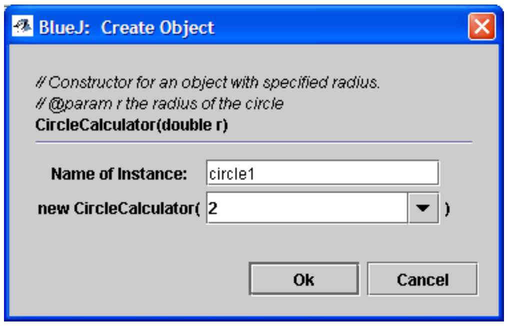
7. Go back to step 1 and create two more objects called `circle2` for a radius of 3, and `circle3` for a radius of 4. You can create as many objects (instances) as you want, each with a different name and radius. Figure 3.4 shows three objects on the object workbench.

8. If you double click on the yellow `CircleCalculator` rectangle an editor window appears showing the Java source code for the class

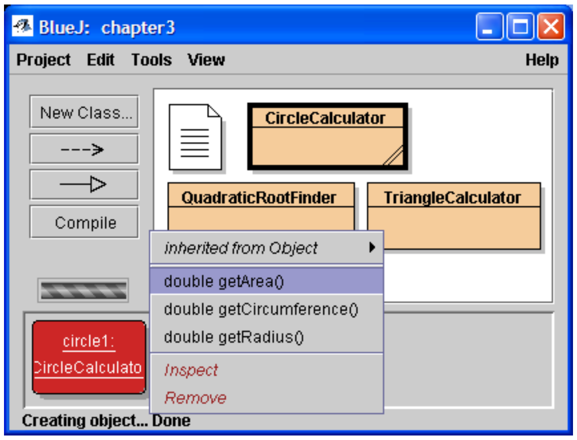
This example elegantly illustrates the three fundamental object-oriented programming (OOP) concepts of **class**, **object**, and **method**.

The class, represented by the yellow rectangle with the class name in it, acts like a blueprint for creating objects. When you double click on it you see the Java code for the class. When you right click on it you can create an object or instance of the class by supplying a name and any arguments.

An object is represented by a red rounded rectangle showing the name of the object and the class that created it. When you right click on an object you can select one of the methods to invoke

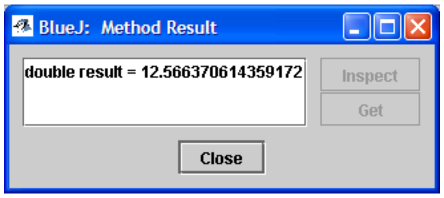

BlueJ: Create Object
// Constructor for an object with specified radius.
// @param r the radius of the circle
CircleCalculator(double r)
Name of Instance: circle1
new CircleCalculator(2)
Ok
Cancel

(a)



(b)

Figure 3.2: (a) shows the dialog box for entering the constructor argument (radius of the circle), and (b) shows how to invoke the getArea method by right clicking on the resulting object.



```
double result = 12.566370614359172
```

Figure 3.3: After choosing the getArea method the result for the area is displayed.

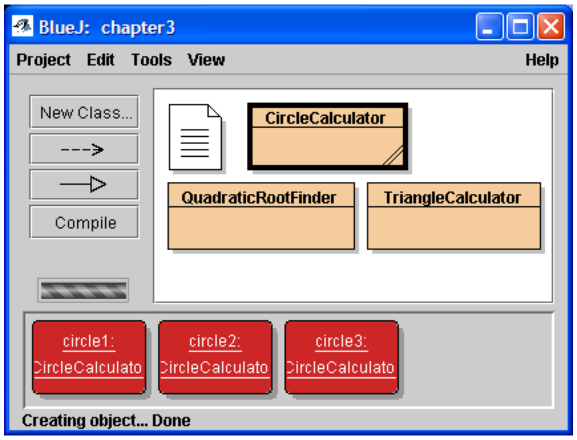


Figure 3.4: Three CircleCalculator objects for radii 2, 3, and 4.

on it. This is sometimes called “sending a message to the object”. This causes the Java code in the class to execute and any results returned by the method are displayed.

We also see that several objects can be constructed from a class. Each has a unique name and its own variables that define the object, such as the radius, area, and circumference.

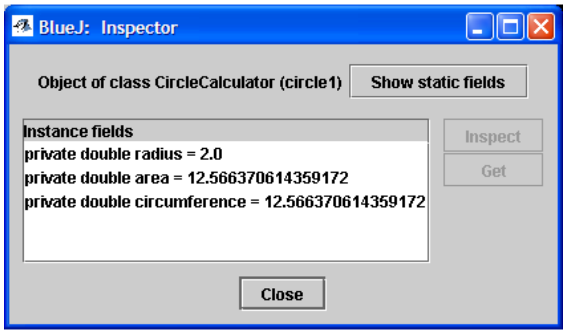
3.2.2 `CircleCalculator` source code

Below we show the source code for the `CircleCalculator` class. For now we have omitted comments so that we can emphasize the structure of the class. Comments are very important for documenting a class and we will show how to include them later in the Chapter. The source code resides in a file called `CircleCalculator.java` and can be viewed in BlueJ by double clicking on the class rectangle.

```
public class CircleCalculator
{
   private double radius;
   private double area;
   private double circumference;

   public CircleCalculator(double r)
   {
      radius = r;
      area = Math.PI * radius * radius;
      circumference = 2.0 * Math.PI * radius;
   }

   public double getRadius()
   {
      return radius;
   }

   public double getArea()
   {
      return area;
   }

   public double getCircumference()
   {
      return circumference;
   }
}
```

3.2.3 Explanation of the source code

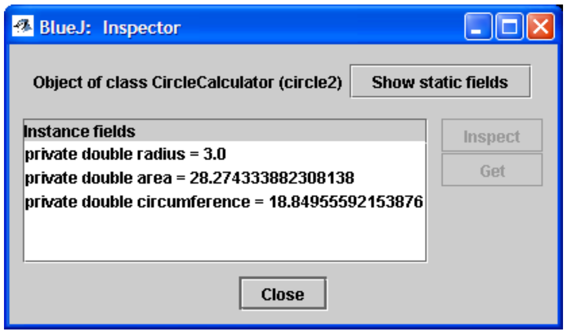
The source code consists of a class declaration containing three parts: (1) instance data fields, (2) constructor declarations, and (3) method declarations.



Object of class CircleCalculator (circle1)

```
Instance fields
private double radius = 2.0
private double area = 12.566370614359172
private double circumference = 12.566370614359172
```

(a)



Object of class CircleCalculator (circle2)

```
Instance fields
private double radius = 3.0
private double area = 28.274333882308138
private double circumference = 18.84955592153876
```

(b)

Figure 3.5: Instance data fields for `circle1` objects in (a) and `circle2` objects in (b).

Class declaration

The following lines are called the class declaration.

```
public class CircleCalculator
{
   ...
}
```

It gives a name to the class and the class definition (**class body**) is contained within the opening and closing braces. A more general template for a class declaration is given in Figure 3.27.

Instance data fields

To define the class we need to specify the variables that uniquely define the state of an object of the class. This is done with the declarations

```
private double radius;
private double area;
private double circumference;
```

The keyword `private` indicates that these variables will not be directly accessible outside the class. Each `CircleCalculator` object will have its own copies of these variables so they are called **instance data fields** or instance variables.

To see this right click on a `CircleCalculator` object and choose inspect on the menu. For example, if we right click on the objects called `circle1` and `circle2` in Figure 3.4 and select inspect the results are shown in Figure 3.5. This shows that each object has its own set of instance data fields. Therefore, an object is often called an **instance** of the class.

Constructor declaration

Next comes the constructor declaration

```
public CircleCalculator(double r)
{
   radius = r;
   area = Math.PI * radius * radius;
   circumference = 2.0 * Math.PI * radius;
}
```

A more general template for a constructor declaration is given in Figure 3.28.

A **constructor** is needed in order to construct an object of the class, so its main purpose is to give values to the instance data fields.

The first line gives the **constructor prototype**. It specifies the name of the constructor, which must always be the same as the name of the class and it specifies what arguments, if any, are necessary to create an object. In our case we only need to specify the radius of the circle as a value of type `double`.

Finally, within the matching braces we place the statements that should be executed when the constructor is used to create an object. These statements are called the **constructor body**.

In Figure 3.2(a) the constructor prototype is displayed and input boxes are available for specifying the object name and the constructor argument `r`. When you fill in these values and click OK the three assignment statements in the constructor are executed to define the object.

It is important to note that the types of the three variables must not be declared in the constructor body since they have already been declared in the instance data field section of the class.

Method declarations

At this stage we have an object on the workbench that is just waiting for something to do. We can tell an object what to do by **invoking** a method on it. These methods are sometimes called **instance methods**. The `CircleCalculator` class contains three such methods, `getRadius`, `getArea`, and `getCircumference`. For example, the `getArea` method declaration is

```
public double getArea()
{
   return area;
}
```

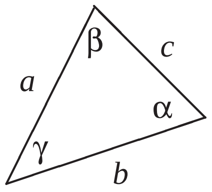
and similarly for the other two methods. A more general template for a method declaration is given in Figure 3.29.

The first line of a method is called the **method prototype**. Our method is public to indicate that it is available outside the class.

Then comes the type of value returned by the method. In our case we are returning the area which has type `double`.

Next comes the name of the method followed by a pair of parentheses which would normally contain any arguments the method requires. The syntax here is the same as for constructor arguments. In our case no arguments are needed but the parentheses are still required

Finally, within the matching braces we place the statements that should be executed when the method is invoked (called). These statements are called the **method body**.



$$c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$$
$$\alpha = \cos^{-1}((b^2 + c^2 - a^2)/(2bc))$$
$$\beta = \cos^{-1}((c^2 + a^2 - b^2)/(2ca))$$
$$perimeter = a + b + c$$
$$s = perimeter/2$$
$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

Figure 3.6: Triangle formulas given two sides *a* and *b*, and the contained angle γ.

When we right click on an object we see a menu of these methods. If we select `getArea` then the statements in its method body are executed. In our case we only need to return the value of one of the instance data fields and that is what the **return** statement does.

A common convention for a method that simply returns the value of an instance data field is to prefix its name with `get`. Such methods are often called **get methods** or **enquiry methods**.

The `CircleCalculator` class is a simple one and you should now understand how the actions performed within BlueJ correspond to the instance data fields, and the execution of code in the bodies of the constructors and methods defined in the class.

3.3 `TriangleCalculator` class using BlueJ

For our second class we will use the formulas for the side lengths and angles of a triangle to solve the following problem:

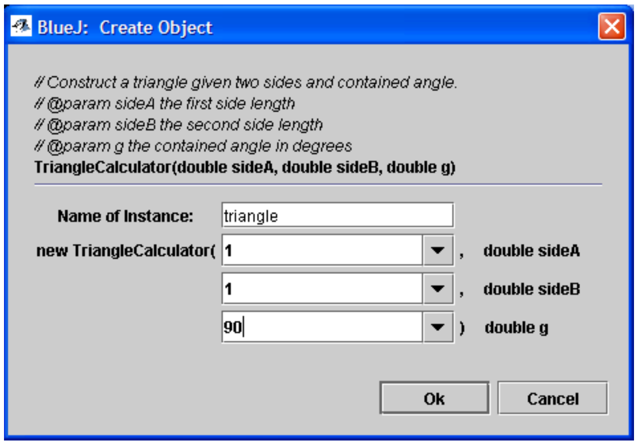
> *"Given the length of two sides of a triangle and the contained angle in degrees, compute the third side length, the other two angles, and the area and perimeter of the triangle."*

If we assume that *a* and *b* are the two side lengths and γ is the contained angle then the formulas are given in Figure 3.6.

3.3.1 Experimenting with the class

We can try out this class by right clicking on the `TriangleCalculator` rectangle to get the "Create Object" dialog box shown in Figure 3.7. We have given our object the name `triangle` and have used the constructor arguments 1, 1, and 90 degrees for the two side lengths and the contained angle.

Right click on the object to get the method menu shown in Figure 3.8. There are nine methods, three return the side lengths, three return the angles, one returns the perimeter, one returns the area, and one checks how close the sum of the angles is to 180 degrees.


BlueJ: Create Object

```
// Construct a triangle given two sides and contained angle.
// @param sideA the first side length
// @param sideB the second side length
// @param g the contained angle in degrees
TriangleCalculator(double sideA, double sideB, double g)
```

Name of Instance: triangle

new TriangleCalculator(1 , double sideA
1 , double sideB
90) double g

Figure 3.7: Dialog box for entering the arguments to construct a `TriangleCalculator` object

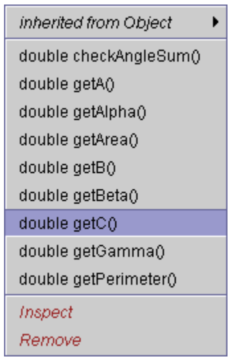
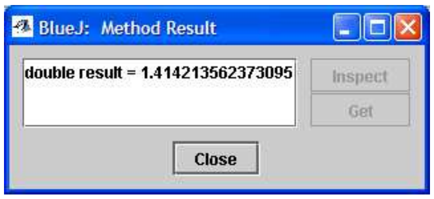
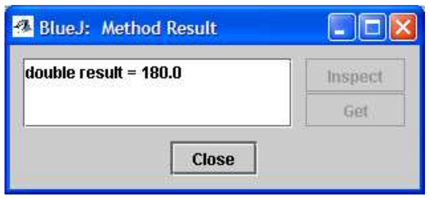


Figure 3.8: The method menu for a `TriangleCalculator` object.



double result = 1.414213562373095

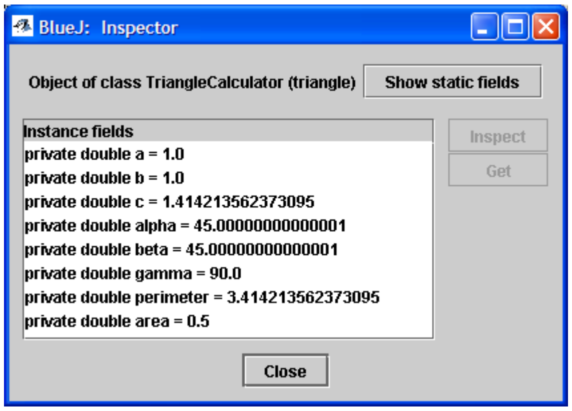
(a)



double result = 180.0

(b)

Figure 3.9: In (a) the value of the third side is shown, and in (b) the sum of the three angles is shown.



Object of class TriangleCalculator (triangle)

```
Instance fields
private double a = 1.0
private double b = 1.0
private double c = 1.414213562373095
private double alpha = 45.00000000000001
private double beta = 45.00000000000001
private double gamma = 90.0
private double perimeter = 3.414213562373095
private double area = 0.5
```

Figure 3.10: The result of choosing inspect from the object menu for a `TriangleCalculator` object.

The results for choosing the `getC` method and the `checkAngleSum` method are shown in Figure 3.9. If we right click on the object again and choose inspect we see the dialog box shown in Figure 3.10. This shows that a `TriangleCalculator` object is defined by eight instance data fields: three sides, three angles, the area, and perimeter.

3.3.2 `TriangleCalculator` source code

Below we show the source code for the `TriangleCalculator` class with the comments omitted for now. The source code resides in a file called `TriangleCalculator.java` and can be viewed in BlueJ by double clicking on the class rectangle.

```
public class TriangleCalculator
{
   private double a, b, c;
   private double alpha;
   private double beta;
   private double gamma;
   private double perimeter, area;

   public TriangleCalculator(double sideA, double sideB, double g)
```

```
{
   double s;

   a = sideA;
   b = sideB;
   c = Math.sqrt(a*a + b*b -2*a*b*Math.cos(Math.toRadians(g)));

   alpha = Math.acos( (b*b + c*c - a*a) / (2*b*c) );
   alpha = Math.toDegrees(alpha);
   beta = Math.acos( (c*c + a*a - b*b) / (2*c*a) );
   beta = Math.toDegrees(beta);
   gamma = g;

   perimeter = a + b + c;
   s = perimeter / 2;
   area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
}

public double getA()
{
   return a;
}

public double getB()
{
   return b;
}

public double getC()
{
   return c;
}

public double getAlpha()
{
   return alpha;
}

public double getBeta()
{
   return beta;
}

public double getGamma()
{
   return gamma;
}

public double getPerimeter()
{
   return perimeter;
}
```

```
   public double getArea()
   {
      return area;
   }

   public double checkAngleSum()
   {
      return alpha + beta + gamma;
   }
}
```

3.3.3 Explanation of the source code

The structure of this class is similar to `CircleCalculator`.

Class declaration

The class declaration is

```
public class TriangleCalculator
{
   ...
}
```

Instance data fields

Inside the class declaration are the declarations for the eight instance data fields that were shown in the “Inspector” window in Figure 3.10:

```
private double a, b, c;
private double alpha;
private double beta;
private double gamma;
private double perimeter, area;
```

Constructor declaration

The constructor declaration has the form

```
public TriangleCalculator(double sideA, double sideB, double g)
{
   ...
}
```

The first line is the constructor prototype, and it indicates that three double arguments are required to construct an object. This constructor corresponds to the “Create Object” dialog box in Figure 3.7.

The constructor body contains the statements that are needed to calculate the third side length, the remaining two angles, the perimeter, and the area using the formulas in Figure 3.6. Nine variables are used here. Eight of them are just the variables that have already been declared as instance data fields. These variables are available anywhere within the class (inside constructor or method bodies).

However, there is also a variable `s` that is just an intermediate variable used only inside the constructor body to simply the calculations so its type must be declared:

```
double s;
```

This is an example of a **local variable**.

Method declarations

The nine methods of this class are shown on the object menu in Figure 3.8. Eight of these methods are just "get methods". Each returns the value of one of the instance data fields.

An additional **enquiry method** with the declaration

```
public double checkAngleSum()
{
   return alpha + beta + gamma;
}
```

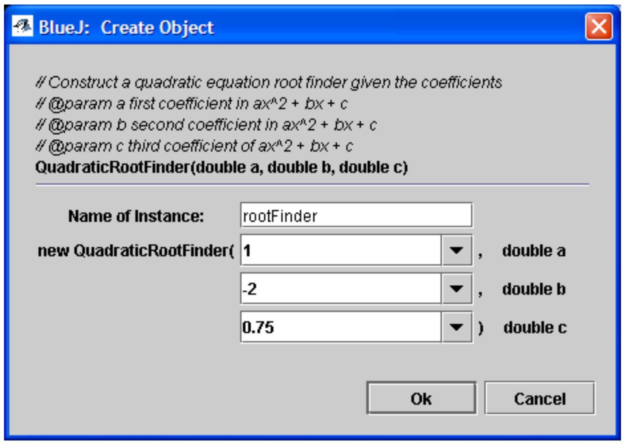
is also included in the class. This method is useful when testing the class.

3.3.4 Testing `TriangleCalculator`

This class is not as simple to test as `CircleCalculator` since it involves some complicated mathematical formulas. If would be easy to make a mistake in the translating of these formulas into Java statements. For example, you could forget that the trigonometric functions require angles in radians instead of degrees, or you could have a plus sign instead of a minus sign, or you could have interchanged *a* and *b* somewhere. Therefore we need to be able to check our results. Here are some ways to do this using BlueJ.

- Use the `checkAngleSum` method. Any significant deviation from 180 degrees means there is some error in the formulas, either the translated ones or the original mathematical ones.

- Develop some test cases for which you know the answer independently. For example, in Figure 3.7, we choose side lengths of 1 and a contained angle of 90 degrees because we know that the third side length is $\sqrt{2} \approx 1.4142$, and the other two angles are 45 degrees. Another simple case is the $30, 60, 90$ triangle with sides 1, $\sqrt{3}$, 2. Other cases could be checked using a calculator.



```
// Construct a quadratic equation root finder given the coefficients
// @param a first coefficient in ax^2 + bx + c
// @param b second coefficient in ax^2 + bx + c
// @param c third coefficient of ax^2 + bx + c
QuadraticRootFinder(double a, double b, double c)
```

Name of Instance: rootFinder

new QuadraticRootFinder(1 , double a

-2 , double b

0.75) double c

Figure 3.11: Dialog box for entering the three arguments to construct a `QuadraticRootFinder` object called `rootFinder`

3.4 `QuadraticRootFinder` class using BlueJ

A quadratic equation has the form $ax^2 + bx + c = 0$ with $a \neq 0$. The values of x which satisfy the equation are called the roots of the equation. The roots r_1 and r_2 are given in terms of a, b, and c by the well-known formulas

$$r_1 = \frac{1}{2a}\left(-b - \sqrt{b^2 - 4ac}\right), \quad r_2 = \frac{1}{2a}\left(-b + \sqrt{b^2 - 4ac}\right)$$

We will assume that the roots are real and we do not check for the square root of a negative number. This requires conditional statements which are introduced in a later Chapter. For testing purposes it is also useful to know that the sum and product of the roots satisfy $r_1 + r_2 = -b/a$ and $r_1 r_2 = c/a$.

3.4.1 Experimenting with the class

We can try out this class by right clicking in the `QuadraticRootFinder` rectangle and selecting the constructor to get the ”Create Object” dialog box shown in Figure 3.11. We have entered values for the quadratic equation $x^2 - 2x + 3/4 = 0$ whose roots are $r_1 = 1/2$ and $r_2 = 3/2$. Right clicking on the object gives the method menu shown in Figure 3.12. There are eight methods associated with a `QuadraticRootFinder` object, five are get methods for returning the coefficients and the roots and three are set methods for modifying the coefficients.

Invoking the `getRoot1` and `getRoot2` methods gives the result boxes shown in Figure 3.13.

Normally, to calculate roots for another quadratic equation we would have to go back and create another object, but this class has “set methods” which can be used to change one or more of the coefficients a, b, or c so it is not necessary to create a new object. If we want to change c to 1 and recalculate the roots we just invoke the `setC` method. This gives the dialog box in Figure 3.14 that prompts for a new value of the argument. Now we can invoke the `getRoot1` and `getRoot2` methods to see the new roots in Figure 3.15. Finally, if we right click on the object and choose inspect we get the “Inspector” window shown in Figure 3.16.

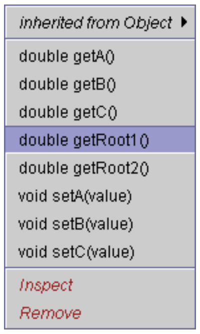
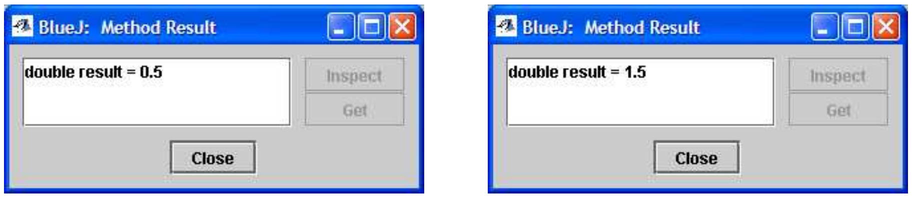


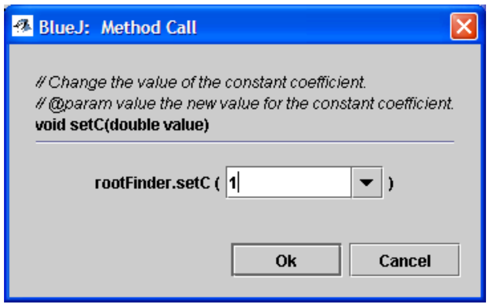
Figure 3.12: The method menu for a `QuadraticRootFinder` object.



double result = 0.5

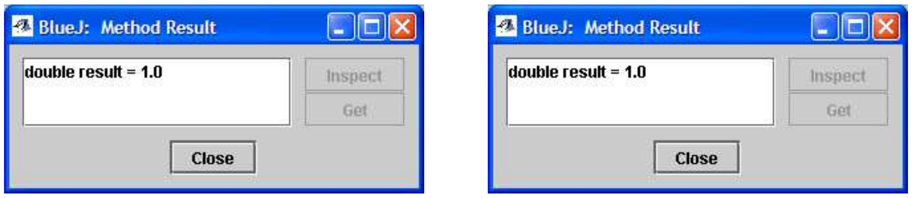
double result = 1.5

Figure 3.13: The roots of the quadratic equation for $a = 1$, $b = -2$ and $c = 3/4$.



```
// Change the value of the constant coefficient.
// @param value the new value for the constant coefficient.
void setC(double value)

rootFinder.setC ( 1 )
```

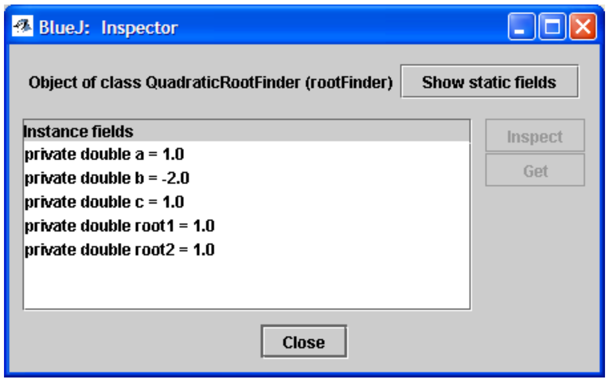
Figure 3.14: The dialog box for the `setC` method.



double result = 1.0

double result = 1.0

Figure 3.15: The roots of the quadratic equation for $a = 1$, $b = -2$, $c = 1$.



Object of class QuadraticRootFinder (rootFinder)

Instance fields
private double a = 1.0
private double b = -2.0
private double c = 1.0
private double root1 = 1.0
private double root2 = 1.0

Figure 3.16: The result of choosing inspect from the object menu for a `QuadraticRootFinder` object corresponding to $a = 1$, $b = -2$ and $c = 1$.

3.4.2 `QuadraticRootFinder` source code

Below we show the source code for the `QuadraticRootFinder` class with the comments omitted for now. The source code resides in a file called `QuadraticRootFinder.java` and can be viewed in BlueJ by double clicking the class rectangle.

```
public class QuadraticRootFinder
{
   private double a, b, c;
   private double root1, root2;

   public QuadraticRootFinder(double a, double b, double c)
   {
      this.a = a;
      this.b = b;
      this.c = c;
      doCalculations();
   }

   private void doCalculations()
   {
      double d = Math.sqrt(b*b - 4*a*c);
      root1 = (-b - d) / (2.0 * a);
      root2 = (-b + d) / (2.0 * a);
   }

   public double getRoot1()
   {
       return root1;
   }

   public double getRoot2()
   {
       return root2;
   }
   public double getA()
   {
      return a;
   }

   public double getB()
   {
      return b;
   }

   public double getC()
   {
      return c;
   }

   public void setA(double value)
   {
      a = value;
      doCalculations();
   }

   public void setB(double value)
   {
      b = value;
      doCalculations();
   }

   public void setC(double value)
   {
      c = value;
      doCalculations();
   }
}
```

3.4.3 Explanation of the source code

There are some new concepts in this class.

Class declaration

The class declaration is

```
public class QuadraticRootFinder
{
   ...
}
```

Instance data fields

The instance data fields are

```
private double a, b, c;
private double root1, root2;
```

and they correspond to the "Inspector" window in Figure 3.16.

Constructor declaration

The constructor declaration is given by

```
public QuadraticRootFinder(double a, double b, double c)
{
   this.a = a;
   this.b = b;
   this.c = c;
   doCalculations();
}
```

In the `CircleCalculator` and `TriangleCalculator` constructors we gave the constructor arguments names that were different from the instance data field names.

Here we give them the same names as the corresponding instance data fields. How do we distinguish between the instance data field names and the argument names? The Java designers thought of this and the answer is to use a special keyword called `this` as a prefix to indicate an instance data field. In the constructor body `this.a` refers to the instance data field variable `a` and `a` refers to the constructor argument whose value is supplied when we create an object by executing the statements in the constructor body.

Method declarations

In the constructor body we find something new, namely the statement

```
doCalculations();
```

When this statement is executed (called) the statements in the `doCalculations` method given by

```
private void doCalculations()
{
   double d = Math.sqrt(b*b - 4*a*c);
   root1 = (-b - d) / (2.0 * a);
   root2 = (-b + d) / (2.0 * a);
}
```

are executed. They calculate the values of the two roots.

The method is declared `private` since it is really just a helper method not needed outside the class. This also means it doesn't appear on the object menu in Figure 3.12.

Our get methods return a value using the `return` statement but the `doCalculations` method doesn't return any value. It just calculates values for the two instance data fields for the two roots. Methods that don't return a value indicate this using the keyword `void` for the return type.

Why do we introduce this method? Why not just use the constructor declaration

```
public QuadraticRootFinder(double a, double b, double c)
{
   this.a = a;
   this.b = b;
   this.c = c;
   double d = Math.sqrt(b*b - 4*a*c);
   root1 = (-b - d) / (2.0 * a);
   root2 = (-b + d) / (2.0 * a);
}
```

The reason can be seen if you notice that the `doCalculations` method is being used in four different places in the class, once in the constructor, and once in each of the three “set” methods. We could have duplicated the three lines of root calculating code in four places but this is not normally good programming practice. Instead we use a technique called **factoring** that replaces repeated blocks of code with a method and uses (calls) the method in several places.

The remainder of the class consists of the five get methods for returning the three coefficients and the two roots, and three set methods. Each **set method** changes one of the instance data fields so it is often called a **mutator method**. This is useful since it means we can solve a new quadratic equation without constructing another object. We simply call the appropriate set methods to change one or more of the coefficients. For example, to change the coefficient `a` we have the method

```
public void setA(double value)
{
   a = value;
   doCalculations();
}
```

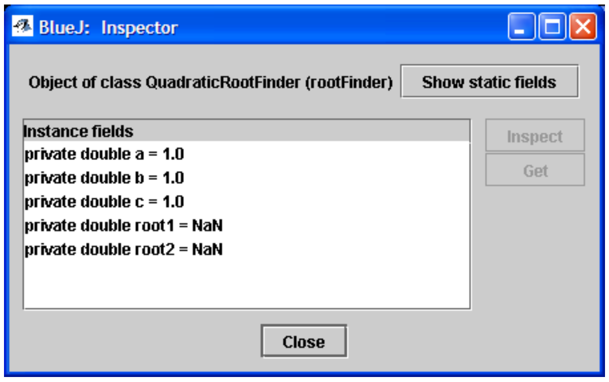
The return type is `void` to indicate that no value is being returned and there is an argument whose value is used to change the value of the instance data field `a`. Since this will change the roots it is necessary to recalculate them by calling the `doCalculations` method. Similar methods are included to change the values of `b` and `c`.

3.4.4 Testing QuadraticRootFinder

To test this class first try some values of a, b, c that give known solutions. For example we have tried the example $x^2 - 2x + 3/4 = (x - 3/2)(x - 1/2) = 0$ and obtained the correct roots $r_1 = 1/2$ and $r_2 = 3/2$. Similarly we tried $x^2 - 2x + 1 = (x - 1)^2 = 0$ and obtained the double root $r_1 = r_2 = 1$.

What happens if you try $x^2 + x + 1 = 0$. In this case there are no real roots since $b^2 - 4ac = -3$. The object inspector gives the results shown in Figure 3.17. This shows that we get the answers `NaN` for both roots. In Java this stands for “not a number” meaning that the result is not a valid `double` number in this case because $\sqrt{-3}$ is not a real number. In fact, we know that the roots are complex numbers in this case.

Another special case to try is $a = 0$, $b = 1$, $c = 1$. In this case the equation is not even a quadratic equation. Nevertheless we get the answers `-Infinity` and `NaN` for the two roots. From the formulas for the roots we see that the first root would be $-2/0$ which is giving `-Infinity` and the second root would be $0/0$ which is giving `NaN`.



```
Object of class QuadraticRootFinder (rootFinder)
Instance fields
private double a = 1.0
private double b = 1.0
private double c = 1.0
private double root1 = NaN
private double root2 = NaN
```

Figure 3.17: The result of choosing inspect from the object menu for a `QuadraticRootFinder` object corresponding to $a = 1$, $b = 1$ and $c = 1$.

Another possible test is to check results using the formulas $r_1 + r_2 = -b/a$ and $r_1 r_2 = c/a$ for the sum and product of the roots.

3.5 Using BeanShell with objects

Within the `BlueJ` environment the creation of objects and the invoking of methods on them is done interactively using the mouse and a dialog box to create the object (for example, Figure 3.1(b) and Figure 3.2(a)) and then selecting the method we want to invoke on the object from the object method menu (for example, Figure 3.2(b)).

Outside the `BlueJ` environment it is necessary to write Java statements to do this. Table 3.1 shows some examples of the correspondence between `BlueJ` mouse actions and menu choices and the Java statements we use outside `BlueJ`.

3.5.1 Constructor call expressions

Table 3.1 shows that to construct a `CircleCalculator` object called `circle1` for a circle of radius 2 we use the statement

```
CircleCalculator circle1 = new CircleCalculator(2.0);
```

The left side of the statement indicates that `circle1` will be the name of an object from the `CircleCalculator` class and the right side uses the keyword `new` to indicate that the constructor should be called to create a new object. The right side of this statement is called a **constructor call expression**.

We can create `TriangleCalculator` and `QuadraticRootFinder` object in a similar way. For example (see Figure 3.7 and Figure 3.11)

```
TriangleCalculator triangle = new TriangleCalculator(1,1,90);
QuadraticRootFinder rootFinder = new QuadraticRootFinder(1,-2,0.75);
```

BlueJ **actions**	**Java statement in** BeanShell

Create a `CircleCalculator` object called `circle1` with radius 2 (Figure 3.1(b) and Figure 3.2(a))	`CircleCalculator circle1 =`
	`     new CircleCalculator(2.0);`

Invoke the `getArea` method on this object (Figure 3.2(b))	`double result = circle1.getArea();`
Seeing the result is automatic (Figure 3.3)	`print(result);`

Construct two more `CircleCalculator` objects (Figure 3.5)	`CircleCalculator circle2 =`
	`     new CircleCalculator(3.0);`
	`CircleCalculator circle3 =`
	`     new CircleCalculator(4.0);`

Choose inspect from object menu to see instance data fields (see Figure 3.5(a))	`print(circle1.getRadius());`
	`print(circle1.getArea());`
	`print(circle1.getCircumference());`

Table 3.1: BlueJ actions and their corresponding Java statements.

In each case the constructor defined in the class declaration is called to create the new object and the left side of the statement gives it a name.

3.5.2 Method call expressions

Now that we have some objects we can invoke methods on them. To do this we need to specify the method name, any required arguments, and the name of the object. For example (see Figure 3.2(a) and Figure 3.3) the statement

```
double result = circle1.getArea();
```

gets the area of the `circle1` object. The right side of this statement is called a **method call expression** and is formed from the object name followed by a dot followed by the method name. We need an empty pair of parentheses here because this method has no arguments. Since we are using a get method, a value is returned and we can assign it to `result`, a `double` variable.

Similarly, we can invoke methods on `TriangleCalculator` and `QuadraticRootFinder` objects. For example (see Figure 3.8 and Figure 3.9)

```
double c = triangle.getC());
double sum = triangle.checkAngleSum();
```

return the third side and angle sum and assign them to variables, and (see Figure 3.13 to Figure 3.15)

```
double r1 = rootFinder.getRoot1();
double r2 = rootFinder.getRoot2();
```

return the two roots of a quadratic equation and assign them to variables.

The statement

```
rootFinder.setC(1.0);
```

invokes the setC method on the QuadraticRootFinder object named rootFinder. It is not an assignment statement since setC does not return a value (its return type is void). It is called an **expression statement**. We know that this method has one argument for the new value of the coefficient c in the quadratic equation so the effect of the method call expression is to changes the value of this coefficient to 1.0.

3.5.3 BeanShell examples

To use BeanShell to experiment with the objects of our three classes we need to tell it where to find the bytecode files that the BlueJ compiler has produced, for example CircleCalculator.class. These are located in the project directory. For example, assuming your project directory for this chapter is

```
c:/book-projects/chapter3
```

then you can type the following command into BeanShell

```
addClassPath("c:/book-projects/chapter3");
```

If you are running BeanShell using Windows then it is important to use forward slashes here instead of backslashes. This change to the classpath remains in effect until you exit BeanShell. The following examples show how BeanShell can be used to construct objects and invoke methods on them.

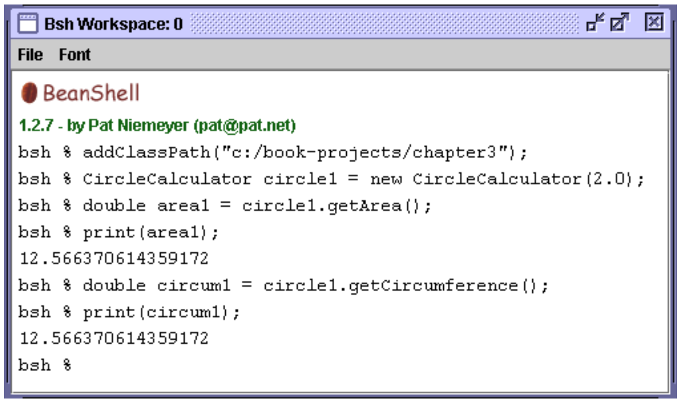
■ EXAMPLE 3.1 **(CircleCalculator objects)** The following statements use BeanShell to calculate the area and circumference of a circle using a CircleCalculator object.

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle1 = new CircleCalculator(2.0);
bsh % double area1 = circle1.getArea();
bsh % print(area1);
12.566370614359172
bsh % double circum1 = circle1.getCircumference();
bsh % print(circum1);
12.566370614359172
```

This example is shown in Figure 3.18 ■

■ EXAMPLE 3.2 **(Three CircleCalculator objects)** Continuing the previous example, the statements

```
bsh % CircleCalculator circle2 = new CircleCalculator(3.0);
bsh % CircleCalculator circle3 = new CircleCalculator(4.0);
bsh % double area2 = circle2.getArea();
bsh % double area3 = circle3.getArea();
bsh % double averageArea = (area1 + area2 + area3) / 3;
bsh % print(averageArea);
30.368728984701335
```



```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle1 = new CircleCalculator(2.0);
bsh % double area1 = circle1.getArea();
bsh % print(area1);
12.566370614359172
bsh % double circum1 = circle1.getCircumference();
bsh % print(circum1);
12.566370614359172
bsh %
```

Figure 3.18: Using BeanShell to construct a `CircleCalculator` object and invoke its methods.

construct two more objects, `circle2` and `circle3`, and compute the average area of the three circles using the `getArea` method. ■

■ EXAMPLE 3.3 **(`TriangleCalculator` objects)** The statements

```
bsh % TriangleCalculator triangle = new TriangleCalculator(1,1,90);
bsh % double c = triangle.getC();
bsh % print(c);
1.414213562373095
bsh % double angleSum = triangle.checkAngleSum();
bsh % print(angleSum);
180.0
```

compute the length of the third side of the right-angled triangle and check the sum of the angles. ■

■ EXAMPLE 3.4 **(`QuadraticRootFinder` objects)** The statements

```
bsh % QuadraticRootFinder rootFinder = new QuadraticRootFinder(1,-2,0.75);
bsh % double r1 = rootFinder.getRoot1();
bsh % double r2 = rootFinder.getRoot2();
bsh % print(r1);
0.5
bsh % print(r2);
1.5
bsh % rootFinder.setC(1);
bsh % r1 = rootFinder.getRoot1();
bsh % r2 = rootFinder.getRoot2();
bsh % print(r1);
1.0
bsh % print(r2);
1.0
```

compute the roots $r_1 = 1/2$ and $r_2 = 3/2$ of the equation $x^2 - 2x + 3/4 = 0$. Then the coefficient c is changed to 1 and the new roots $r_1 = r_2 = 1$ are computed. ■

3.6 Writing and viewing Javadoc class documentation

The three classes in this chapter have been presented without any comments. It is essential that every class you write include comments to document the purpose of the class, its constructors and methods. There are three kinds of comments in Java:

Single line comments If you use `//` then these characters and all others following them on the same line are ignored by the Java compiler. For example in the `TriangleCalculator` class we can use

```
private double gamma;   // angle opposite side c
```

to indicate the purpose of the variable `gamma`.

Multi-line comments They begin with `/*` on one line and end on the same line or a following one with the characters `*/`. For example, the private method in `QuadraticRootFinder` can be documented as follows:

```
/* This private method is used in the constructor and the
 * three set methods in order to update the roots in case
 * a coefficient is changed.
 */
private void doCalculations()
{
   double d = Math.sqrt(b*b - 4*a*c);
   root1 = (-b - d) / (2.0 * a);
   root2 = (-b + d) / (2.0 * a);
}
```

The extra asterisks on the two intermediate lines are not necessary but they are supplied by the BlueJ editor automatically so we will use them.

Javadoc comments They look like regular multi-line comments but they begin with `/**` so that the javadoc processor can identify them. Of course the Java compiler sees them as ordinary multi-line comments and ignores them. For example, here is a javadoc version for the `CircleCalculator` constructor.

```
/** Constructor for an object with specified radius.
 *  @param r the radius of the circle
 */
public CircleCalculator(double r)
{
        radius = r;
        area = Math.PI * radius * radius;
        circumference = 2.0 * Math.PI * radius;
    }
```

Within a javadoc comment HTML tags can be used and there are special tags beginning with the `@` character. In this example we have used the `@param` tag to describe the constructor argument. The HTML and special tags are used to apply special formatting to the class documentation (interface). The resulting HTML document can be viewed by a browser.

The javadoc comments are also shown by `BlueJ` in the "Create Object" dialog boxes.

3.6.1 Javadoc rules

First we need some of the javadoc rules:

- Use a javadoc block comment immediately before the class declaration to give a description of the class. This comment can contain the special `@author` and `@version` tags.

- Use a javadoc comment immediately before each public constructor and method declaration. The first line (ended by first period) is special and is used in the summary part of the documentation. Any remaining lines give further information that is shown in the detail part of the documentation.

- Use a parameter line for each argument. It has the format

  `@param` *name text*.

  where *name* is the name of the argument and *text* describes the argument.

- Use a return line for each method that returns a value. It has the format

  `@return` *text*.

There are other javadoc tags that we won't need yet. We can now give the complete javadoc versions of our three classes.

3.6.2 Javadoc version of `CircleCalculator`

Class `CircleCalculator`

book-projects/chapter3

```
package chapter3; // remove this line if you are not using packages
/**
 * The objects of this class know how to compute the area and
 * circumference of a circle, given its radius as a constructor
 * argument (parameter).
 */
public class CircleCalculator
```

```
{
   // instance data fields defining a CircleCalculator object

   private double radius;
   private double area;
   private double circumference;

   /** Constructor for an object with specified radius.
    *  @param r the radius of the circle
    */
   public CircleCalculator(double r)
   {
      radius = r;
      area = Math.PI * radius * radius;
      circumference = 2.0 * Math.PI * radius;
   }

   /**
    * Return the radius of the circle.
    * @return circle radius
    */
   public double getRadius()
   {
      return radius;
   }

   /**
    * Return the area of the circle.
    * @return circle area
    */
   public double getArea()
   {
      return area;
   }

   /**
    * Return the circumference of the circle.
    * @return circle circumference
    */
   public double getCircumference()
   {
      return circumference;
   }
}
```

3.6.3 Javadoc version of `TriangleCalculator`

Class `TriangleCalculator`

book-projects/chapter3

```
package chapter3; // remove this line if you are not using packages
```

```
/**
 * A TriangleCalculator represents a triangle by two side lengths and
 * the contained angle in degrees. From this information the third
 * side length and remaining two angles can be calculated. Then the
 * area and perimeter can be calculated. All values can be returned
 * using get methods.
 */
public class TriangleCalculator
{
   private double a, b, c; // triangle side lengths
   private double alpha;   // angle opposite side a
   private double beta;    // angle opposite side b
   private double gamma;   // angle opposite side c

   private double perimeter, area;

   /**
    * Construct a triangle given two sides and contained angle.
    * @param sideA the first side length
    * @param sideB the second side length
    * @param g the contained angle in degrees
    */
   public TriangleCalculator(double sideA, double sideB, double g)
   {
      double s;

      a = sideA;
      b = sideB;
      c = Math.sqrt(a*a + b*b -2*a*b*Math.cos(Math.toRadians(g)));

      // Angle opposite side a, contained by sides b and c

      alpha = Math.acos( (b*b + c*c - a*a) / (2*b*c) );
      alpha = Math.toDegrees(alpha);

      // Angle opposite side b, contained by sides c and a

      beta = Math.acos( (c*c + a*a - b*b) / (2*c*a) );
      beta = Math.toDegrees(beta);

      gamma = g;

      // Calculate perimeter and use Heron’s formula for
      // the area in terms of the side lengths

      perimeter = a + b + c;
      s = perimeter / 2;
      area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
   }

   /**
    * Return the length of side a.
```

```
     * @return the length of side a
     */
    public double getA()
    {
       return a;
    }

    /**
     * Return the length of side b.
     * @return the length of side b
     */
    public double getB()
    {
       return b;
    }

    /**
     * Return the length of side c.
     * @return the length of side c
     */
    public double getC()
    {
       return c;
    }

    /**
     * Return the angle opposite side a.
     * @return the angle opposite side a in degrees
     */
    public double getAlpha()
    {
       return alpha;
    }

    /**
     * Return the angle opposite side b.
     * @return the angle opposite side b in degrees
     */
    public double getBeta()
    {
       return beta;
    }

    /**
     * Return the angle opposite side c.
     * @return the angle opposite side c in degrees
     */
    public double getGamma()
    {
       return gamma;
    }
```

```
   /**
    * Return the perimeter of the triangle.
    * @return the perimeter of the triangle
    */
   public double getPerimeter()
   {
      return perimeter;
   }

   /**
    * Return the area of the triangle
    * @return the area of the triangle
    */
   public double getArea()
   {
      return area;
   }

   /**
    * Return the sum of the angles as a check that it is close
    * to 180 degrees.
    * @return the sum of the angles
    */
   public double checkAngleSum()
   {
      return alpha + beta + gamma;
   }
}
```

3.6.4 Javadoc version of `QuadraticRootFinder`

Class `QuadraticRootFinder`

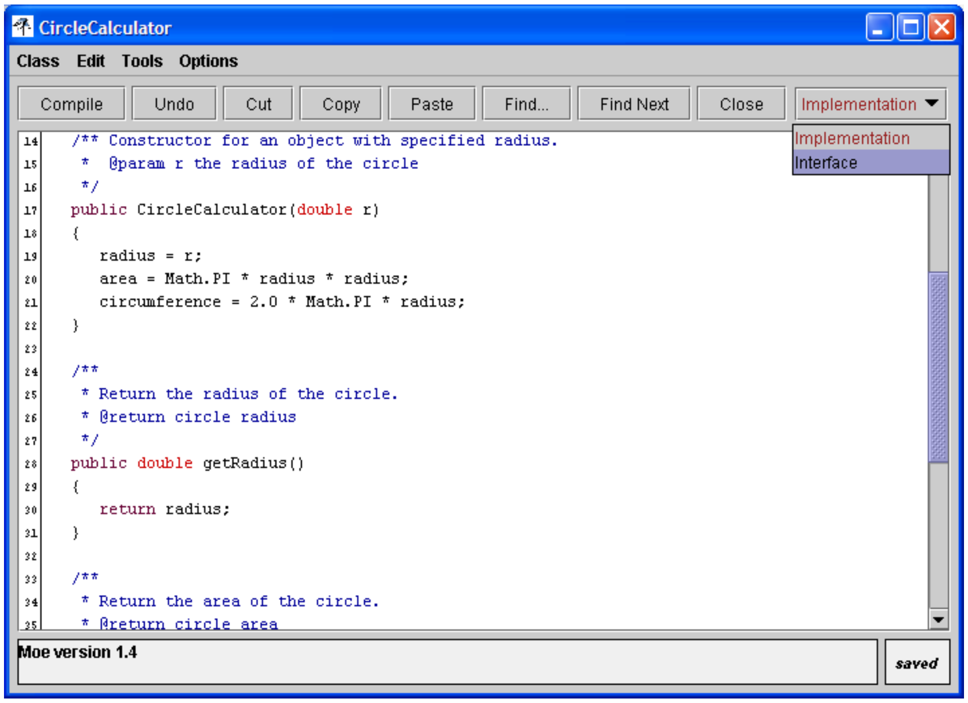
book-projects/chapter3

```
package chapter3; // remove this line if you are not using packages
/**
 * An object of this class can calculate the real roots of the
 * quadratic equation ax^2 + bx + c = 0 given the coefficients a, b, and c.
 * The program does not check if there are real roots. Later when
 * we know how to make decisions (if statements) we can make a better
 * version of this class.
 */
public class QuadraticRootFinder
{
   // Instance data fields for coefficients and roots

   private double a, b, c;
   private double root1, root2;

   /**
    * Construct a quadratic equation root finder given the coefficients
```

```
     * @param a first coefficient in axˆ2 + bx + c
     * @param b second coefficient in axˆ2 + bx + c
     * @param c third coefficient of axˆ2 + bx + c
     */
    public QuadraticRootFinder(double a, double b, double c)
    {
       this.a = a;
       this.b = b;
       this.c = c;
       doCalculations();
    }

    /* This private method is used in the constructor and the
     * three set methods in order to update the roots in case
     * a coefficient is changed.
     */
    private void doCalculations()
    {
       double d = Math.sqrt(b*b - 4*a*c);
       root1 = (-b - d) / (2.0 * a);
       root2 = (-b + d) / (2.0 * a);
    }

    /**
     * Return the first root.
     * @return the first real root or NaN if there are none
     */
    public double getRoot1()
    {
        return root1;
    }

    /**
     * Return the second real root.
     * @return the second real root or NaN if there are none
     */
    public double getRoot2()
    {
        return root2;
    }

    /**
     * Return the coefficient of xˆ2.
     * @return the coefficient of xˆ2
     */
    public double getA()
    {
       return a;
    }

    /**
     * Return the coefficient of x.
     * @return the coefficient of x
     */
    public double getB()
    {
       return b;
    }

    /**
     * Return the constant coefficient.
     * @return the constant coefficient
     */
    public double getC()
    {
       return c;
    }

    /**
     * Change the value of the coefficient of xˆ2.
     * @param value the new value for the coefficient of xˆ2
     */
    public void setA(double value)
    {
       a = value;
       doCalculations();
    }

    /**
     * Change the value of the coefficient of x.
     * @param value the new value for the coefficient of x
     */
    public void setB(double value)
    {
       b = value;
       doCalculations();
    }

    /**
     * Change the value of the constant coefficient.
     * @param value the new value for the constant coefficient.
     */
    public void setC(double value)
    {
       c = value;
       doCalculations();
    }
}
```

3.6.5 Viewing the documentation

With BlueJ it is very easy to generate and display the Java documentation. For example, to generate documentation for the `CircleCalculator` class double click on its rectangle to bring up the



```
/** Constructor for an object with specified radius.
 *  @param r the radius of the circle
 */
public CircleCalculator(double r)
{
   radius = r;
   area = Math.PI * radius * radius;
   circumference = 2.0 * Math.PI * radius;
}

/**
 * Return the radius of the circle.
 * @return circle radius
 */
public double getRadius()
{
   return radius;
}

/**
 * Return the area of the circle.
 * @return circle area
```

Figure 3.19: An editor window.

editor window as shown in Figure 3.19. Now select the implementation button menu in the top right corner of the toolbar and select interface. The documentation will be generated in the editor window. Part of it is shown in Figure 3.20. You can use this button to toggle between the source code (implementation) and the interface (documentation).

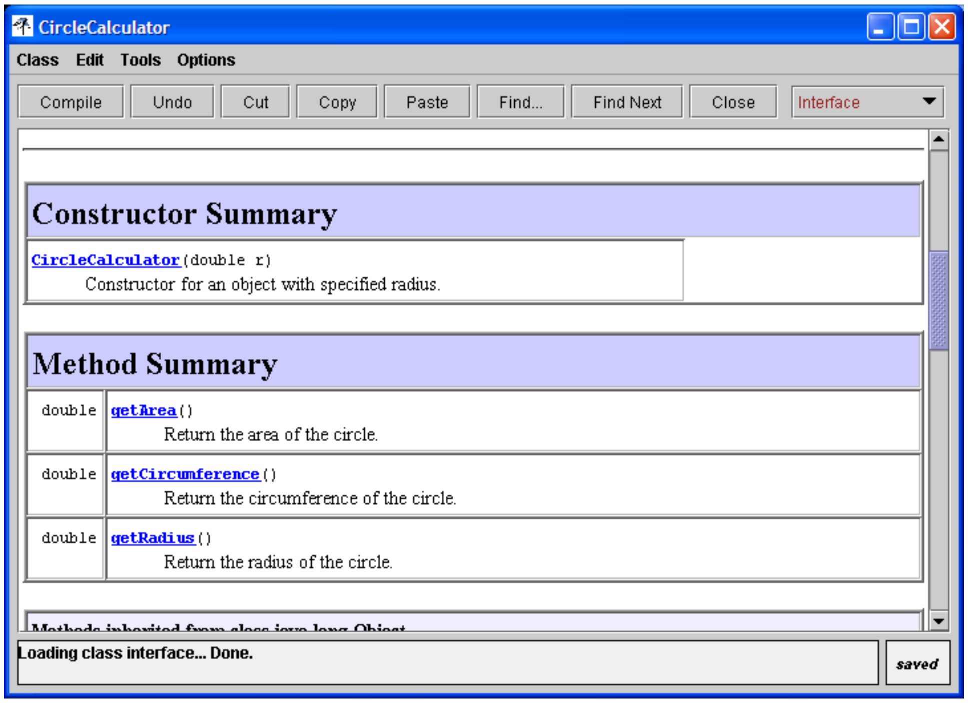
3.6.6 Implementation and documentation views

The **implementation** and documentation give two different views of a class. The implementation gives the complete view of the source code including all comments, all constructor and method bodies. The documentation produced by javadoc is often called the **public interface** or **specification** of the class. It includes only the javadoc comments, the public class, constructor, and method prototypes (first lines) but not the method bodies, all nicely formatted as an HTML document. Private data fields and methods are not shown in the documentation.

These two views relate to how we use the class. As a programmer writing Java classes we are writing the complete source code (the implementation) but as someone that is simply using the class, as we did in our BlueJ experiments, it is only necessary to view the public interface. It gives all the information needed to use the class.

3.6.7 Project documentation

Normally a project contains more than one class. The project for this chapter contains three classes. It is possible to generate the Java documentation for all classes in a project simultaneously by


CircleCalculator
Class Edit Tools Options
Compile Undo Cut Copy Paste Find... Find Next Close Interface
saved

Constructor Summary

CircleCalculator(double r)
Constructor for an object with specified radius.

Method Summary

double	getArea() Return the area of the circle.
double	getCircumference() Return the circumference of the circle.
double	getRadius() Return the radius of the circle.

Methods inherited from class java.lang.Object

Loading class interface... Done.

Figure 3.20: Generated Java documentation obtained from the implementation/interface button.

choosing "Project Documentation" from the BlueJ "Tools" menu. The documentation will appear in your browser instead of the editor window. The first page of the results is shown in Figure 3.21.

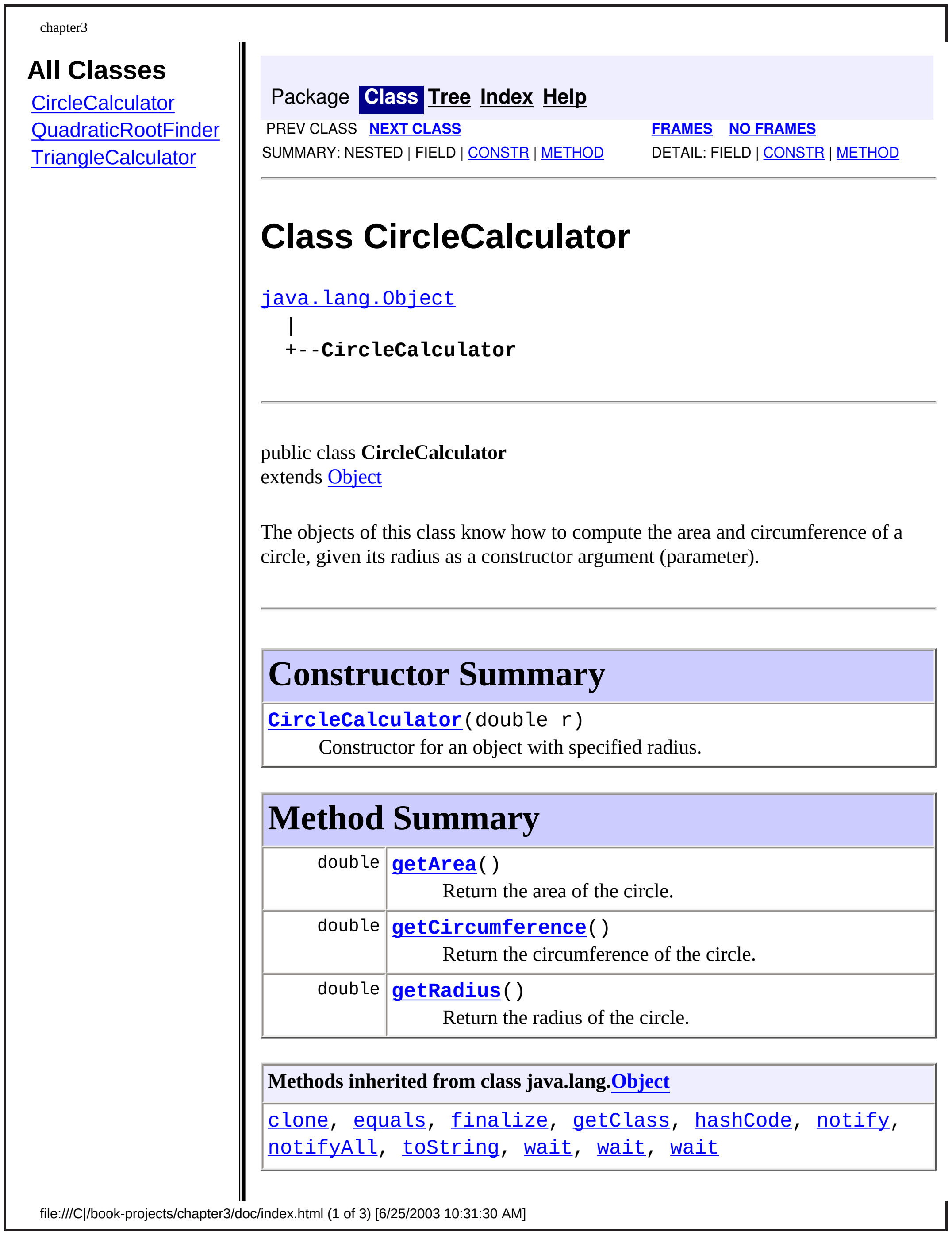
3.7 Syntax and logical errors

When writing and testing a Java class it is rare that your first attempt is without error so it is important to recognize errors and be able to fix them.

There are two kinds of errors that can occur: **syntax errors** and **logical errors**. The Java language is defined by a number of syntax or grammar rules that are used by the compiler to determine whether a Java statement is legal or not. If a statement is illegal we say that it contains one or more syntax errors. These errors are often called **compile-time** errors since they are found by the compiler when it attempts to compile your class to obtain the bytecode file. Forgetting the semi-colon at the end of a statement is a common example of a syntax error.

Logical errors are often called **run-time errors** since they occur when the Java interpreter is executing your class. A logical error may result in an abnormal termination of execution of a constructor or method in your class or it may simply produce erroneous results because you used a minus sign instead of a plus sign in some formula.

Finding logical errors can be difficult and can be accomplished only by thoroughly testing your classes. Therefore we will place a lot of emphasis on techniques for testing classes. BlueJ is an excellent environment for testing.


chapter3

All Classes

CircleCalculator
QuadraticRootFinder
TriangleCalculator

Package **Class** **Tree** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class CircleCalculator

```
java.lang.Object
  |
  +--CircleCalculator
```

public class **CircleCalculator**
extends Object

The objects of this class know how to compute the area and circumference of a circle, given its radius as a constructor argument (parameter).

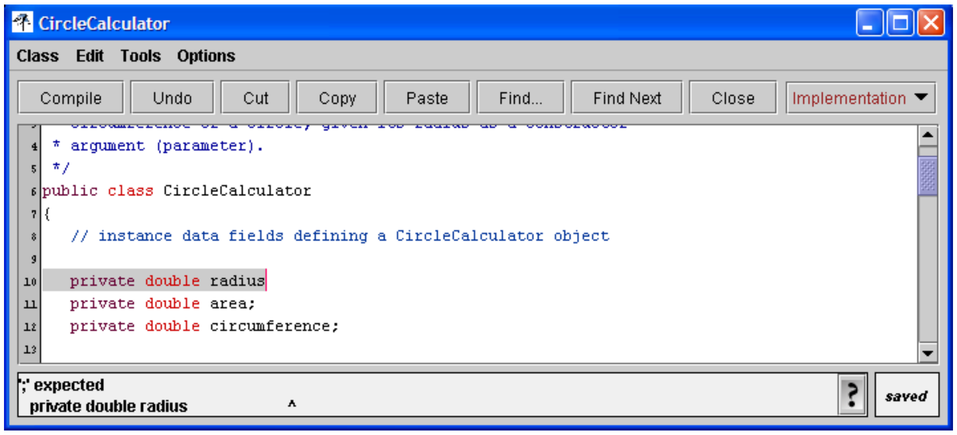
Constructor Summary
`CircleCalculator(double r)`   Constructor for an object with specified radius.

Method Summary	
`double`	`getArea()` Return the area of the circle.
`double`	`getCircumference()` Return the circumference of the circle.
`double`	`getRadius()` Return the radius of the circle.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

file:///C|/book-projects/chapter3/doc/index.html (1 of 3) [6/25/2003 10:31:30 AM]

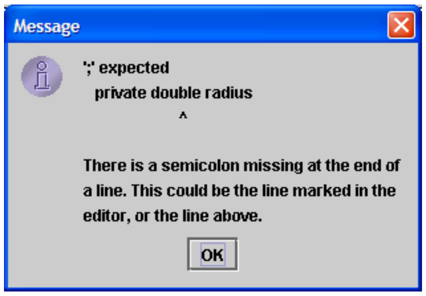
Figure 3.21: A web page for the chapter3 project documentation


CircleCalculator
Class Edit Tools Options
Compile Undo Cut Copy Paste Find... Find Next Close Implementation
?
saved

```
 * argument (parameter).
 */
public class CircleCalculator
{
   // instance data fields defining a CircleCalculator object

   private double radius
   private double area;
   private double circumference;

```

```
';' expected
 private double radius
                      ^
```

Figure 3.22: BlueJ syntax error for a missing semi-colon.



```
';' expected
 private double radius
                      ^
```

There is a semicolon missing at the end of a line. This could be the line marked in the editor, or the line above.

Figure 3.23: Detailed BlueJ error message for a missing semi-colon.

3.7.1 Some common syntax errors

Forgetting a semi-colon

For example, open the editor for the `CircleCalculator` class and remove the semi-colon from the end of the line

```
public double radius;
```

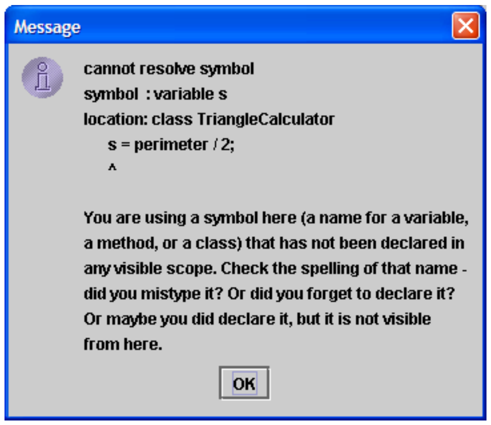
Now compile the class using the Compile button on the editor toolbar and you will get an error message at the bottom of the editor window as shown in Figure 3.22. Click on the question mark button to see the more detailed message shown in Figure 3.23.

Undeclared variables

In the `TriangleCalculator` class it would be easy to omit the line

```
double s;
```

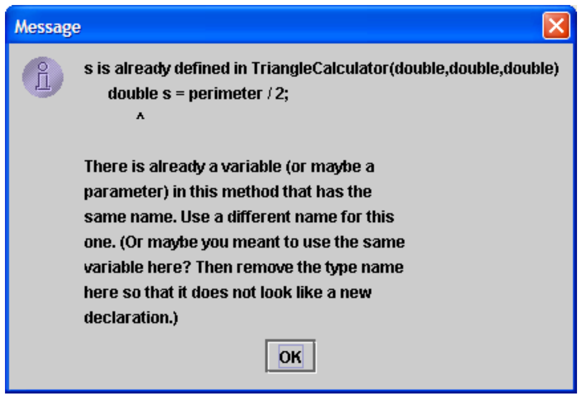
in the constructor body. Then you would get the error message shown in Figure 3.24



```
cannot resolve symbol
symbol  : variable s
location: class TriangleCalculator
      s = perimeter / 2;
      ^
```

You are using a symbol here (a name for a variable, a method, or a class) that has not been declared in any visible scope. Check the spelling of that name - did you mistype it? Or did you forget to declare it? Or maybe you did declare it, but it is not visible from here.

Figure 3.24: Detailed BlueJ error message for an undeclared variable.



```
s is already defined in TriangleCalculator(double,double,double)
      double s = perimeter / 2;
             ^
```

There is already a variable (or maybe a parameter) in this method that has the same name. Use a different name for this one. (Or maybe you meant to use the same variable here? Then remove the type name here so that it does not look like a new declaration.)

Figure 3.25: Detailed BlueJ error message for a variable declared twice.

Declaring a variable more than once

In the `TriangleCalculator` constructor declaration the local variable `s` is declared for the first time using

```
double s;
```

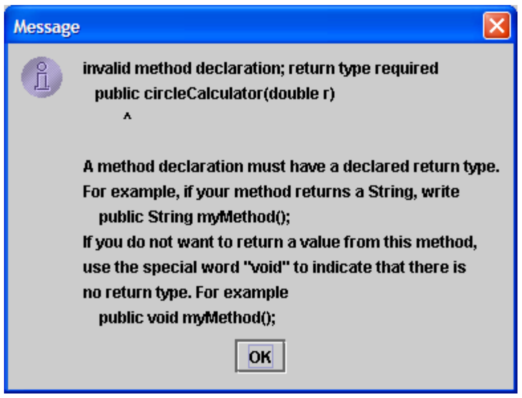
Later it is used in the line

```
s = perimeter / 2;
```

Replace this line by

```
double s = perimeter / 2;
```

Compile the class and you will get the error message shown in Figure 3.25.



```
invalid method declaration; return type required
  public circleCalculator(double r)
         ^

A method declaration must have a declared return type.
For example, if your method returns a String, write
   public String myMethod();
If you do not want to return a value from this method,
use the special word "void" to indicate that there is
no return type. For example
   public void myMethod();
```

Figure 3.26: Detailed BlueJ error message for misspelled constructor name.

Misspelling the constructor name

For example, in the `CircleCalculator` class (page 63) suppose the constructor declaration is written as

```
public circleCalculator(double r)
{
   ...
}
```

We have used a lowercase `c` instead of an uppercase `C`. If we compile the class we get the syntax error shown in Figure 3.26. The error message indicates that the compiler is trying to consider `circleCalculator` as a method but it doesn't have a return type. The compiler does not consider this as a constructor declaration because constructors must have the same name as the class.

Forgetting `new` in constructor call expressions

We can't illustrate this error in BlueJ yet, but it is easy to make in some of our BeanShell examples. For example, try the following statements.

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = CircleCalculator(2.0);
// Error: Typed variable declaration : Command not found: CircleCalculator:
<at unknown location>
```

Here we forgot to use `new` on the right hand side of the statement. The error message is not so helpful here. Without `new` the Java interpreter is assuming that the right hand side is a method call (like `getArea()`) but there is no method with this name.

3.7.2 Some common logical errors

There are several very common logical errors that programmers make in Java.

Using an incorrect formula

As an example consider the `TriangleCalculator` formulas in Figure 3.6 that are implemented in the constructor (page 64). Any number of logical errors could be made here such as using a `+` sign instead of a `-` sign. Errors like this can be discovered by testing.

Redeclaring an instance variable

This is difficult error for beginners to detect. For example, in the `CircleCalculator` class (page 63) if we had replaced the assignment statements

```
radius = r;
area = Math.PI * radius * radius;
circumference = 2.0 * Math.PI * radius;
```

in the constructor body by the declarations

```
double radius = r;
double area = Math.PI * radius * radius;
double circumference = 2.0 * Math.PI * radius;
```

then the class would compile just fine; there are no syntax errors. However, when you construct an object and ask it for the radius, area, and circumference, you will get `0.0` as an answer for any radius. By declaring the variables in the constructor we have introduced three local variables that have nothing to do with the instance data fields of the same name so the assignment of values to them does not change the instance data field values.

For numeric instance data fields the compiler will automatically initialize their values to zero and they will remain zero since we didn't initialize them in the constructor. The local variables disappear when the constructor body finishes execution. Therefore, when you invoke the `getRadius`, `getArea`, and `getCircumference` methods on an object, `0.0` is returned in each case. This declaration of a variable in a constructor or method having the same name as an instance variable is called **shadowing** and should be avoided.

Using a return type on a constructor

This is also a common error that is easy to make and is difficult for a beginner to find and understand.

Suppose the `CircleCalculator` constructor (page 63) is replaced by

```
public void CircleCalculator(double r)
{
   ...
}
```

We have erroneously used `void` in the constructor prototype. This is not a syntax error. The compiler simply assumes that `CircleCalculator` is the name of a method, not the name of a constructor.

If you try this in BlueJ by right clicking on the class you will see `new CircleCalculator()` and you will not be asked to enter a radius to construct an object. When you construct an object and right click to get its menu of methods you will see `void CircleCalculator(r)` there, indicating that this is a method.

What has happened is that by using a return type we have in effect not used any constructor in our class. When the compiler notices this it automatically provides a so-called default do-nothing constructor having the form

```
public CircleCalculator()
{
}
```

with no arguments and an empty body. So this is what you are using to create an object and again when you invoke the `getRadius`, `getArea`, and `getCircumference` methods on an object, `0.0` is returned in each case.

3.7.3 Invoking a method on a non-existent object

In our BeanShell examples we could have constructed our objects in two steps as we did sometimes for `int` and `double` variables: first declare them and later assign values to them. For example we could use the statements

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle;
bsh % circle = new CircleCalculator(2.0);
bsh % double area = circle.getArea();
bsh % print(area);
12.566370614359172
bsh % circle = new CircleCalculator(3.0);
bsh % area = circle.getArea();
bsh % print(area);
28.274333882308138
bsh %
```

The first statement declares `circle` to be a variable of type `CircleCalculator` and the second statement constructs an object to assign to it.

However suppose the second statement was accidentally omitted. Then the third statement is trying to invoke the `getArea` method on a non-existent object. In BeanShell we would get

```
bsh % CircleCalculator circle;
bsh % double area = circle.getArea();
// Error: // Uncaught Exception: Typed variable declaration :
Null Pointer in Method Invocation: <at unknown location>
Target exception: java.lang.NullPointerException
```

This cryptic error message tells us that we have a variable `circle` of type `CircleCalculator` but we have not constructed an object to assign to it. Later you will better understand this error message.

3.8 Summary of terminology

In this section we give simple definitions of the important terms introduced in this Chapter. Many terms are used to discuss and explain a language such as Java. It is important that you understand them and can give examples of each term.

For example, there are general language-independent terms such as **variable** that would be used in any computer language, and terms such as **class**, **object**, and **method** that would be used in any object-oriented language.

Other terms and definitions would be specific to Java. For example, the Java language is defined by a set of rules called the **grammar** or **syntax** of the language. Although these rules can be formally defined we will introduce them in simple informal way.

Simple identifier

A sequence of one or more letters, digits and underscores such that the first character is not a digit. Identifiers are used to give names to variables, classes, constructors, objects, and methods.

An almost universal convention is to begin the name of a class with an upper case letter. All other identifiers begin with a lower case letter. In either case capitalize the beginning letter of each interior word.

Identifiers are **case sensitive**.

Example: `radius`, `numberOfStudents` are variable names
Example: `CircleCalculator` is a class name
Example: `doCalculations`, `checkAngleSum` are method names
Example: `circle1`, `triangle` are object names.

type

A specific kind of data such as the set of all integers or the set of all real numbers, or the set of all `CircleCalculator` objects.

Example: `int`, `float`, `double` are primitive types.
Example: `CircleCalculator` is an object type.

class (definition 1)

A definition of a set of objects of a specific type and their behavior.

Example: `CircleCalculator`, `TriangleCalculator`

class (definition 2)

A home for some functions not associated with any objects.

Example: `Math` is our only example so far.

object

An entity that has identity (a name), state, and behavior.

Example: A `CircleCalculator` object.

instance

An object constructed from a class.

Example: A `CircleCalculator` object that has the name `circle1` is an instance of the `CircleCalculator` class.

method

A function or operation defined in a class that can be invoked on an object of the class. Such methods are often called **instance methods**. Later we will learn that there are also **static methods**. The instance methods of a class define the behavior of objects.

Example: `getArea`, `setA`, `checkAngleSum`, `doCalculations`

constructor

A special kind of method defined in a class that is used to create an object (instance of the class) having specified properties. An object must be created before any of its methods can be invoked (executed). A constructor must have the same name as its class.

Example: `QuadraticRootFinder`

variable declaration

A statement having one of the forms

accessModifier typeName identifier`;`
accessModifier typeName identifier `=` *expression*`;`

where, for now, *accessModifier* is either absent or `private`, *identifier* is the name of the variable, *typeName* is the variable type and *expression* is an expression that evaluates to a value that can be assigned to the variable.

Example: `double radius;`
Example: `double radius = 2.0;`
Example: `double area = Math.PI * radius * radius;`
Example: `CircleCalculator circle1 = new CircleCalculator(2.0);`
Example: `CircleCalculator circle1;`
Example: `int n=123, remainder, hundreds, tens, units;`
Example: `double area, circumference;`
Example: `double radius = 3.0, area;`

The final three examples show that multiple variables of the same type can be declared and optionally initialized in a single declaration.

constant declaration

A constant has the form

accessModifier `static final` *typeName* *identifier* `=` *expression*`;` The *accessModifier* can be `public` or `private`. The strange keyword `static` indicates that constants are associated with the class, not the objects of the class The equally strange keyword `final` distinguishes a constant declaration from an initialized variable declaration.

It is conventional to name constants using upper case letters and the underscore to simulate a space.

Example: `static final double CM_PER_INCH = 2.54;`

arithmetic expression

An expression involving variables and operators that evaluates to a numeric value.

Example: `radius`
Example: `2.0 * Math.PI * radius`
Example: `remainder % 10`
Example: `circle1.getArea() + circle2.getArea()`

assignment statement

A statement of the form

identifier `=` *expression* `;`

where *identifier* is the name of a variable that has already been declared and *expression* is an expression whose value is assigned to the variable.

Example: `radius = 2.0;`
Example: `circle1 = new CircleCalculator(2.0);`
Example: `area = Math.PI * radius * radius;`

class declaration

A template for a simple class declaration is shown in Figure 3.27. The first box is replaced by the name of the class. The other boxes show that a class declaration has three parts, not all of which are required in every class. These parts are defined below.

Example:

```
public class CircleCalculator
{
   ...
}
```

The part indicated by `{...}` is called the **class body**.

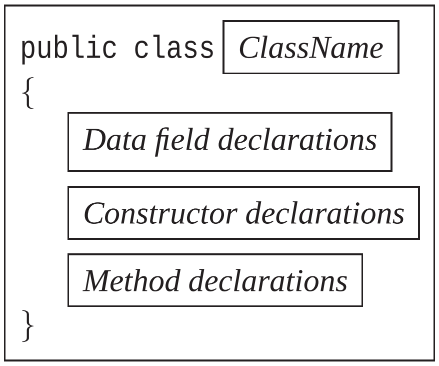


Figure 3.27: A template for a simple Java class declaration.

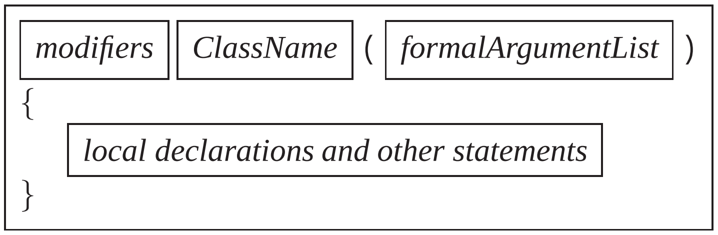


Figure 3.28: A template for a simple Java constructor declaration.

instance data field

An instance data field is a special variable declaration in the body of a class but outside any method or constructor. These variables are available anywhere in the class and are the only variable declarations that have a modifier such as `private`. Each object of the class has its own copies of the instance data fields.

Example: `private double radius;`
Example: `private double root1, root2;`

The second example shows that more than one variable can be declared in one declaration.

constructor declaration

A template for a constructor declaration is shown in Figure 3.28. The *modifiers* box can be replaced by `public` which is the only modifier we have discussed so far. Then we have the name of the class followed by a formal argument list in parentheses, if any. The *formalArgumentList* is is list of type-variable pairs separated by commas.

Example:

```
public TriangleCalculator(double sideA, double sideB, double g)
{
```

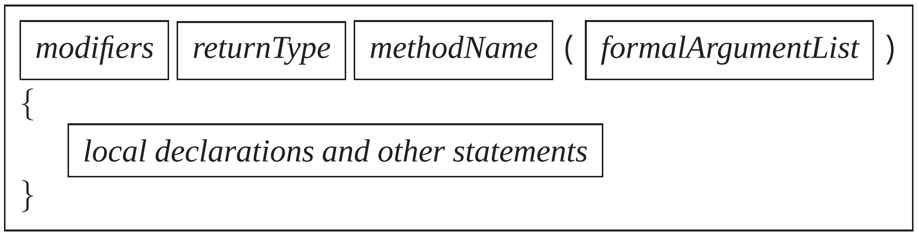


Figure 3.29: A template for a simple Java method declaration.

```
        double s;
        a = sideA;
        ...
        area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
    }
```

The part indicated by `{...}` is called the **constructor body**.

constructor prototype

The first line of a constructor declaration is the constructor prototype.

Example: `public CircleCalculator(double r)`
Example: `public TriangleCalculator(double sideA,`
```
                               double sideB, double g)
```

For a constructor, only the prototype is part of the public interface (see Javadoc output).

constructor call expression

An expression of the form

`new` *ClassName*`(`*actualArguments*`)`

that constructs an object from the class whose name is *ClassName* and whose *actualArguments*, if any, is a list of expressions separated by commas that evaluate to a value of the type indicated in the *formalArgument* list of the constructor prototype.

Example: `new CircleCalculator(3.0)`
Example: `new QuadraticRootFinder(1.0,1.0,90.0)`

method declaration

A template for a method declaration is shown in Figure 3.29. The *modifiers* box can be replaced by `public` or `private` which are the only modifiers we have discussed so far. Then we have *returnType*, the name of the return type, followed by the name of the method, followed by a formal argument list in parentheses, if any. The *formalArgumentList* is is list of type-variable pairs separated by commas. The difference between a constructor and a method is that a method always has a return type and a name that begins with a lowercase letter and a constructor has no return type and a name that begins with an uppercase letter.

Example:

```
public double getArea()
{
   return area;
}
```

Example:

```
public void setC(double value)
{
   c = value;
   doCalculations();
}
```

Example:

```
private void doCalculations()
{
   double d = Math.sqrt(b*b - 4*a*c);
   root1 = (-b - d) / (2.0 * a);
   root2 = (-b + d) / (2.0 * a);
}
```

The part of a method declaration indicated by `{...}` is called the **method body**.

method prototype

The first line of a method declaration is the method prototype.

Example: `public double getArea()`
Example: `public void setC(double value)`
Example: `public void doCalculations()`

For a method, only the prototype is part of the public interface (see Javadoc output).

method call expression

An expression of one of the forms

objectName`.`*methodName*`(`*actualArguments*`)`
methodName`(`*actualArguments*`)`

The first form invokes a method called *methodName* on an object called *objectName*. The *actualArguments*, if any, is a list of expressions separated by commas that evaluate to a value of the type indicated in the formal argument list of the method prototype. The second form, without an object name, is used to invoke a method in the same class in which the method is defined.

Example: `circle1.getArea()`
Example: `rootFinder.setC(1.0)`
Example: `doCalculations()`

enquiry method

A method that returns information specific to an object of a class, without changing the object.

Example: In the `TriangleCalculator` class the `checkAngleSum` method returns the sum of the three angles.

mutator method

A method that changes the state of an object of a class, usually by modifying one or more instance data fields.

Example: `doCalculations` in the `QuadraticRootFinder` class calculates new values for the instance variables `root1` and `root2`.

get method

A special kind of enquiry method whose name is `get`*Name* where *name* is the name of one of the instance data fields. Its purpose is to return the value of this field.

Example: `getArea` returns the value of instance variable `area` in `CircleCalculator`.

set method

A special kind of mutator method whose name is `set`*Name* where *name* is the name of one of the instance data fields. Its purpose is to change the value of this field.

Example: In the `QuadraticRootFinder` class `setC` modifies the value of the instance variable `c`.

local variable

A local variable is quite different from an instance variable. An instance variable is declared in a class outside any constructor or method and it can be used inside any constructor or method of the class.

A local variable is declared in the body of a constructor or method and cannot be used outside the constructor or method. It comes into existence each time the body is executed and disappears when the constructor or method finishes execution.

Example: `double s;` declares a local variable in the `TriangleCalculator` constructor.

formal argument

A formal argument in a method or constructor is a special local variable whose value is supplied when a method or constructor call expression is executed (called).

Example: In the `setC` method of the `QuadraticCalculator` class, with prototype `void setC(double value)`, `value` is a local variable of type `double`.

actual argument

An actual argument is a variable or expression whose value is used as the value of the corresponding formal argument when a method or constructor call expression is executed (called).

Example: For the `setC` method of the `QuadraticRootFinder` class, with prototype `void setC(double value)`, the expression `rootFinder.setC(1.0)` causes the value `1.0` to be assigned as the value of the formal argument `value`.

`return` statement

A return statement has two forms:

```
return expression;
return;
```

The first form is used to indicate that the value of *expression* is to be returned by the method. The second form is used in a method that has `void` return type. We havn’t seen an example of this form yet. When a `return` statement is executed the method finishes execution immediately.

Example: `return alpha + beta + gamma;`

single line comment

A comment beginning with `//`. These characters and all following characters on the same line are part of the comment.

multi-line comment

A comment beginning with `/*` and ending with `*/` on the same or another line.

javadoc comment

A comment beginning with `/**` and ending with `*/` on the same or another line.

3.9 Review exercises

► **Review Exercise 3.1** Define the following terms and give examples of each.

simple identifier	type	class
object	instance	method
constructor	variable declaration	constant declaration
class declaration	instance data field	constructor declaration
constructor prototype	constructor call expression	method declaration
method prototype	method call expression	enquiry method
“get” method	mutator method	“set” method
local variable	formal argument	actual argument
`return` statement	single line comment	multi-line comment
javadoc comment	`@param`	`@return`
addClassPath	public interface	class specification
class implementation	syntax error	logical error
run-time error	undeclared variable error	duplicate definition error
redeclaration error		

► **Review Exercise 3.2** Make a list of all the prototypes for the constructors used in this chapter.

► **Review Exercise 3.3** Make a list of all the constructor call expressions used in this chapter.

► **Review Exercise 3.4** Make a list of all the prototypes for the methods used in this chapter.

► **Review Exercise 3.5** Make a list of all the method call expressions used in this chapter.

3.10 Programming exercises

In each programming exercise you should include javadoc comments and indicate what data you have used to test your class.

► **Exercise 3.1 (Converting inches to feet and inches)**
Write a class called `InchesToFeetConverter` whose constructor argument is an integer representing the height of a person in inches. This number is to be converted to feet and inches. For example for a height of 67 inches is 5 feet and 7 inches. Use the following class outline and fill in the details indicated by `{..}`.

```
public class InchesToFeetConverter
{
   private int heightInInches;
```

```
      private int feetPart;
      private int inchesPart;

      public InchesToFeetConverter(int height) { ... }
      public int getHeightInInches() { ... }
      public int getFeetPart() { ... }
      public int getInchesPart() { ... }
   }
```

► Exercise 3.2 (Height in metric units)
Write a class called `HeightConverter` whose constructor has two integer arguments for the feet part and the inches part of a height. This height is to be converted into centimeters. Use constants for the conversion factors from centimeters to inches (2.54) and from feet to inches (12.0). For example, for a height of 5 feet 10 inches the height in centimeters is 177.8. Use the following class outline and fill in the details indicated by `{..}`.

```
public class HeightConverter
{
   // put your constants here

   private double heightCM;

   public HeightConverter(int feet, int inches) { ... }
   public double getHeightCM() { ... }
}
```

► Exercise 3.3 (Fahrenheit to Celsius temperature conversion)
Write a class called `FToCConverter` that can be used to convert a Fahrenheit temperature to a Celsius temperature. The constructor needs one argument for the given temperature in Fahrenheit. Do not include any “set methods” in your class.

► Exercise 3.4 (Celsius to Fahrenheit temperature conversion)
Write a class called `CToFConverter` that can be used to convert a Celsius temperature to a Fahrenheit temperature. The constructor needs one argument for the given temperature in Celsius. Do not include any “set” methods in your class.

► Exercise 3.5 (Heat loss and windchill calculator)
Write a class called `WindChillCalculator` with a constructor that has two `double` arguments. One us for the temperature in degrees Celsius and the other is the wind speed in kilometers per hour. The constructor should use the formulas in Example 2.22 and Example 2.23 to do the calculations. To use the heat loss formula in Example 2.23 you will have to convert the input wind speed from kilometers per hour to meters per second. As an example if the wind speed is 30 km/hr and the temperature is -15C then the windchill is approximately -33.8 and the heat loss is approximately 1487.2. Include set methods for the temperature and for the wind speed.

► Exercise 3.6 (Making change)
Write a class called `ChangeHelper` that helps a cashier give change to a customer. The constructor has two inputs, (1) the amount due as a `double` value (e.g., 3.28 is 3 dollars and 28 cents) and the amount received as a `double` value (e.g., 5.00 is 5 dollars and 0 cents). Also assume that the amount received from the customer is equal to or greater than the amount due. The constructor should calculate the minimum number of dollars, quarters, dimes, nickels, and cents the customer should receive as change. Each of these values should be returned using a "get" method. HINT: First convert the two double numbers to total pennies (multiply by 100), round to the nearest integer, subtract and use `/` and `%` a few times to extract the numbers of each type of coin.

► Exercise 3.7 (Calculating Easter)
The day and month on which Easter falls can be calculated using quotients and remainders with the following steps involving fifteen variables (a to p) starting with the value of y for the given year.

Step	Dividend	Divisor	Quotient	Remainder
1	y	19	–	a
2	y	100	b	c
3	b	4	d	e
4	$8b+13$	25	f	–
5	$11(b-d-f)-4$	30	g	–
6	$7a+g+6$	11	h	–
7	$19a+(b-d-f)+15-h$	29	–	i
8	c	4	j	k
9	$(32+2e)+2j-k-i$	7	–	m
10	$90+(i+m)$	25	n	–
11	$19+(i+m)+n$	32	–	p

The value of n is the month number (3 for March, 4 for April) and the value of p is the day of the month. Write an `EasterCalculator` class whose constructor has one integer argument for the year. Provide "get methods" for the month number and the day number for Easter.

► Exercise 3.8 (An interesting formula for the Fibonacci numbers)
The Fibonacci numbers ($F_0, F_1, F_2, \ldots$) occur often in computer science. They are integers and the sequence beginning with F_0 can be calculated exactly using the recurrence relation $F_n = F_{n-1} + Fn - 2$ where $F_0 = F_1 = 1$. Later when we learn about loops you can use this formula to calculate them exactly.

However there is an interesting closed formula for the nth Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

This is a strange formula since it appears that the results are not necessarily integers for all $n = 0, 1, 2, \ldots$. However, it can be shown that they are all integers (the $\sqrt{5}$ factors all cancel out).

Write a class called `FibonacciCalculator` with a constructor taking one integer argument for the value of n. Provide a "get" method to return the value of F_n using `double` calculations with the formula. For example, some exact values are $F_{10} = 55$, $F_{20} = 675$, $F_{30} = 832040$, and $F_{40} = 102334155$ but your program will only give the approximate result $F_{30} = 832040.0000000008$. Why are the answers not quite integers?

What is the largest F_k that can be calculated exactly by truncating the floating point result?

► **Exercise 3.9 (Calculating e^x)**
A series representing e^x is given by

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

in the sense that, for a given x, using more terms gives better approximations to e^x. Write a class called `ExpCalculator` with a constructor having one `double` argument for the value of x and two "get" methods. One returns the accurate value of e^x obtained using `Math.exp(x)` and the other returns the approximate value obtained using this series up to the term in x^4. Do not use the `Math.pow` method to compute the powers. Instead write the approximation in the form

$$e^x = 1 + x(1 + x(1/2 + x(1/6 + x(1/24))))$$

Use your class to discover the range of x values for which the approximation agrees with the more exact value to at least 5 significant figures.

For example, when x is 0.1 the accurate value is 1.1051709180756477 and the value from the series is 1.1051708333333332. showing that for $x = 0.1$ the two results agree to at least 5 significant figures (in fact, 7 significant figures).

Chapter 4

Classes, Objects, and Methods

OOP Concepts

Outline

String class

Writing classes that use strings

Writing classes that use association and aggregation

Using other library classes

Object construction

Object references

Data encapsulation and integrity

Running a class from the command line

4.1 Introduction

In this Chapter we continue with the basic concepts and terminology of object-oriented programming (OOP). One of the main themes is that existing library classes and classes we write ourselves can be used together as building blocks to create the classes we need to solve problems.

First we introduce the `String` class which represents sequences of characters and we write some classes that use the `String` class. We introduce other important library classes such as the `Date`, `Calendar`, `SimpleDateFormat` and `NumberFormat` classes.

The important OOP concepts of association and aggregation are explained using several examples, including a `Point` class and a `Circle` class.

Then we summarize and extend the important OOP concepts we have encountered so far. These include object construction, object references, comparison of primitive and object types, comparison of assignment statements for primitive and reference types, using references as arguments and method return values, data encapsulation and integrity, side-effects (both desirable and undesirable), comparison of instance and static variables and methods, and the call by value argument passing mechanism use by Java.

Other useful library classes such as the `Calendar`, `Date`, and `SimpleDateFormat` are also introduced.

We write several classes that show how to use the `String` class and that also illustrate the important ideas of association and aggregation. The BlueJ environment is very useful for exploring these concepts.

Finally, we introduce the `main` method which is necessary in order to run Java classes outside BlueJ from the command line.

4.2 `String` class

In Java the `String` class is part of a standard Java library (package) called `java.lang` and contains many methods for operating on string objects. Each `String` object is a sequence of 0 or more characters. This is one of the most important classes since strings are used in almost every program.

We will explore this class using BeanShell and BlueJ and use strings in our own classes just like we used the primitive `int` and `double` types in Chapter 3.

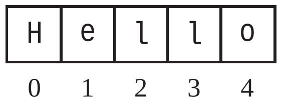
4.2.1 Constructing strings

A **literal string** is a sequence of characters delimited by double quote characters. For example, `"Hello"` is a literal string. The double quote characters are used to delimit the characters included in the string but are not part of the string. The length of a string is defined as the number of characters in the string, so this string has length 5.

The **empty string** has no characters and a length of zero. It is denoted by `""`, two consecutive double quote characters with nothing between them.

Each character in a string is a Unicode character of type `char` stored internally as a 16-bit integer code. The `char` data type was briefly discussed in Chapter 2. To use strings we normally do not need to understand how characters are represented internally as integers.

Each character in a string can be directly referenced by an index. The index begins at zero so the first character has index 0, the next has index 1, and so on. The string `"Hello"` can be represented by the picture



Here each character is shown in a box and its index is shown below the box. The index can be used to specify individual characters in the string. In the picture the first `l` has index 2 (or position 2).

A **substring** of a given string is a string constructed from a subsequence of characters from the string beginning at one index position and ending at another. In the picture the substring `"ell"` begins at index 1 and ends at index 3.

To construct the literal string object `"Hello Fred"` and give it the name `greeting` we use the declaration statement

```
String greeting = "Hello Fred";
```

Strings are special, we do not need to use `new` to create a `String` object as we did in Chapter 3 (see Table 3.1 for example).

It is very important to understand that `greeting` is defined on the left side of this declaration as the name of a `String` object, not the `String` object itself. We say that `greeting` is an **object reference variable**. The right side creates the `String` object and returns a reference to it which is then assigned to `greeting`. Loosely speaking we often say that `greeting` is an object when we really mean that it is an object reference variable.

A picture of this process is shown in Figure 4.1. The box named `greeting` holds the reference,

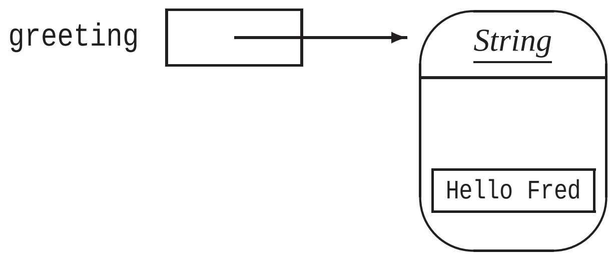


Figure 4.1: A string object in memory and a reference to it called `greeting`

shown as an arrow, and the rounded box represents the memory reserved for the object and the characters in the string. This is very different from the way the primitive types such as `int` and `double` work (compare with Figure 2.2)

4.2.2 String expressions and concatenation

The most common string operation is **concatenation**. It takes two strings and joins them together to make a new string. It is also called **append** since it appends one string to the end of another. In

Java the + operator is used to denote string concatenation and it should not be confused with its use in the addition of numbers. If `s1`, `s2`, ..., `sn` are either strings or expressions that can be converted to strings then

```
s1 + s2 + ... + sn
```

is called a string expression and it is always evaluated from left to right unless there are parentheses. We can use BeanShell to illustrate string construction and concatenation

■ EXAMPLE 4.1 **(Constructing literal strings)** The statements

```
bsh % String name = "William " + "James " + "Duncan";
bsh % print(name);
William James Duncan
```

show how to concatenate three literal strings to make a new string. ■

■ EXAMPLE 4.2 **(String variable expressions)** Generalizing the preceding example, define the string variables `first`, `middle` and `last` for three names and concatenate them into a full name using the statements

```
bsh % String first = "William";
bsh % String middle = "James";
bsh % String last = "Duncan";
bsh % String fullName = first + " " + middle + " " + last;
bsh % print(fullName);
William James Duncan
bsh %
```

The last declaration concatenates five strings, two of which are strings consisting of a single space. The three variables are replaced by their string values. ■

String expressions containing numbers

String expressions that contain both strings and numbers are very convenient. They are called mixed string expressions and are evaluated like pure string expressions except that any numbers or other expressions are converted to strings before the results are concatenated together.

■ EXAMPLE 4.3 **(Mixed string expressions)** In the BeanShell statements

```
bsh % String area = "Area: " + Math.sqrt(2.0);
bsh % print(area);
Area: 1.4142135623730951
```

the square root function returns a `double` value which is converted to a string and then concatenated with the literal string. ■

There are some pitfalls to watch out for, as the following example shows.

■ EXAMPLE 4.4 **(Parentheses in string expressions)** Suppose that `x` and `y` are integer variables that have the values `3` and `4`, respectively. The evaluation of the string expression

```
"The sum of " + x + " and " + y + " is " + x + y
```

gives the string expression

```
"The sum of 3 and 4 is 34"
```

which is probably not what you want. To obtain the desired result

```
"The sum of 3 and 4 is 7"
```

it is necessary to use parentheses as in

```
"The sum of " + x + " and " + y + " is " + (x + y)
```

so that the final `+` is interpreted as an addition instead of a string concatenation. ■

4.2.3 String methods

Many methods are available in the `String` class. For example, we can find the length of a string, the character at a specified position, or a substring. As we did in Chapter 2, for the functions in the `Math` class, we can document the string methods using their prototypes.

The length of a string

The prototype for the length method is

```
public int length()
```

indicating that this is an instance method that returns the number of characters in the string. Since string indices begin at zero, a valid index `i` for a string `s` should be in the range

$$0 \le \texttt{i} \le \texttt{s.length()} - 1$$

■ EXAMPLE 4.5 **(Length of a string)** The BeanShell statements

```
bsh % String name = "Harry";
bsh % int len = name.length();
bsh % print(len);
5
```

use the instance method call expression `name.length()` to assign the length of `name` to the integer variable `len`. ■

Converting a number to a string

■ EXAMPLE 4.6 **(Converting numbers to strings)** If `age` is an integer variable and `area` is a double precision variable then the statements

```
String s1 = "" + age;
String s2 = "" + area;
```

convert the numbers to strings by using the empty string to force `+` to be interpreted as concatenation. This trick forces numeric values to be converted to strings. ■

Extracting a single character from a string

Sometimes we need to extract specific characters from a string. There is an instance method called `charAt` to do this: It has the prototype

```
public char charAt(int index)
```

The return value is the character at the index (beginning at 0) as specified by the argument.

■ EXAMPLE 4.7 **(Extracting a character from a string)** The statements

```
bsh % String s = "Hello";
bsh % char c = s.charAt(1);
bsh % print(c);
e
bsh % print(s.charAt(5));
// Error: // Uncaught Exception: ...
... java.lang.StringIndexOutOfBoundsException: String index out of range: 5
```

show how to assign the character at position 1 in the string `"Hello"` to a `char` value and display it. The last statement shows that if you specify an index outside the valid range (0 to 4 here), a `StringIndexOutOfBoundsException` error message is displayed. ■

Constructing a substring

The `substring` method constructs a new string that is a substring of a given string. There are two versions with prototypes

```
public String substring(int firstIndex)
public String substring(int firstIndex, int lastIndexPlusOne)
```

These instance methods are used to send messages to a string object and return one of its substrings. The first version has one formal argument and returns the substring beginning at index `firstIndex` and continuing to the end of the string. The second version has two formal arguments. It returns the substring starting at index `firstIndex` and ending at index `lastIndexPlusOne - 1` rather than the index of the last character of the substring as you might expect.

Since the `String` class is immutable the original string is unchanged. The substring operation creates a new string.

■ EXAMPLE 4.8 **(Substrings)** Let `d` be an integer variable with a value in the range 1000 to 999999. The BeanShell statements

```
bsh % int d = 531452;
bsh % String sd = "" + d;
bsh % int len = sd.length();
bsh % sd = "$" + sd.substring(0,len-3) + "," + sd.substring(len-3);
bsh % print(sd);
$531,452
```

convert it to a string in the range `$1,000` to `$999,999`. The second `substring` expression extracts the digits that go after the comma and the first extracts the digits that precede the comma. ■

Trimming a string

The `trim` method can be used to remove leading and trailing spaces from a string. It is an instance method with the prototype

```
public String trim()
```

A new string that has no leading or trailing spaces is created by this method and a reference to it is returned.

■ EXAMPLE 4.9 **(Trimming a string)** The BeanShell statements

```
bsh % show();
<true>
bsh % String s, t;
bsh % s = "      Hello       ";
<      Hello       >
bsh % t = s.trim();
<Hello>
```

create `t`, a trimmed version of `s`. ■

Upper case and lower case conversions

There are methods to convert strings from lower case to upper case and vice versa. The prototypes are

```
public String toLowerCase()
public String toUpperCase()
```

The first method can be used to construct a lowercase version of a string with all uppercase letters replaced by lowercase ones, and the second method does the opposite.

■ EXAMPLE 4.10 **(Case conversion)** The BeanShell statements

```
bsh % String test = "Hello";
bsh % String upper = test.toUpperCase();
bsh % print(upper);
HELLO
```

creates an upper case version of the string `test`. Similarly, the statements

```
bsh % String test = "transfer";
bsh % char first = test.toUpperCase().charAt(0);
bsh % print(first);
T
```

return the upper case version of the first character of the string `test`. ■

Searching for substrings

Sometimes it is useful to know if one string is a substring of another one. There are four instance methods to do this. Their prototype are

```
public int indexOf(int ch)
public int indexOf(int ch, int startIndex)
public int indexOf(String sub)
public int indexOf(String sub, int startIndex)
```

The first two methods search for a character `ch` in a string and the last two methods search for a string `sub`. The one argument versions begin the search at the start of the string and the two argument versions begin at `startIndex`.

In any case the value returned is -1 if the character or substring is not found. Otherwise the index of the character or the first character of the substring is returned.

Also notice that these four methods all have the same name. This is an example of **method overloading**. The compiler can determine which version to use by looking at the **signature** of the method: the types and number of the arguments.

■ EXAMPLE 4.11 **(Searching using `indexOf`)** The BeanShell statements

```
bsh % String indices = "0123456789012345678901234567890";
bsh % String target  = "This is the target string";
bsh % print(target.indexOf('u'));
-1
bsh % print(target.indexOf('g'));
15
bsh % print(target.indexOf("the"));
8
bsh % print(target.indexOf("target",12));
12
bsh % print(target.indexOf("target",13));
-1
bsh %
```

show how to search for characters and substrings. ■

4.2.4 Displaying numbers and strings

In BeanShell we have been using either `show` or `print` to display numbers and strings. In BlueJ we have dialog boxes that appear and show the result (return value) of invoking a method that returns a value.

These display techniques are not really part of Java. However Java can produce what is called **console output** or **terminal output**, if you have a console or terminal window, using an object called `System.out` which is automatically provided by the Java interpreter. It has two methods called `print` and `println` for displaying data.

Some of the method prototypes are

```
public void println(int n)
public void println(double d)
public void println(String s)
public void println()
```

Here `println` stands for "print line". There are many different `println` methods having different argument types. We have shown four common ones here. This is another example of method overloading. In each case the value of the actual argument is converted to a string and displayed on a line in the terminal window and the next output will begin on a new line. The `println` method with no argument simply moves to the next line.

For each `println` method there is a corresponding `print` method. The difference is that after a `print` method is executed the next output will begin on the same line.

Using `System.out` in BeanShell

In BeanShell if you select "Capture System in/out/err" from the File menu and then you can use `System.out`.

■ EXAMPLE 4.12 **(Using `System.out` in BeanShell)** Try the statements

```
double area = 3.14159;
System.out.println("Area: " + area);
System.out.print("Area: ");
System.out.println(area);
```

in the BeanShell editor and see the results in the workspace. The last two statements produce the same output as the first `println` statement. ■

Special characters in strings

The `println` statement

```
System.out.println("Hello");
```

has the same effect as the `print` statement

```
System.out.print("Hello\n");
```

The backslash used in a string is called an **escape character** and its use affects the meaning of the next character. In this example it means to interpret `n` as the "newline" character instead of its literal meaning. Thus, the newline character can be expressed as `'\n'`. It is most useful in print statements since each time it is used it causes a line break to the beginning of the next line.

There are several of these special characters. For example, the carriage return character is denoted by `'\r'` and the tab character is denoted by `'\t'`.

If you need to include a backslash literally in a string then it is necessary to use two backslashes like this: `\\`. Or, you may want to use the double quote character literally in a string. This can be done using `\"` in the string to specify that the double quote character is not the string delimiter.

■ EXAMPLE 4.13 **(Using backslash as an escape character)** The statement

```
System.out.println("\"\\Hello\\\"");
```

displays the string

```
"\Hello\"
```

since two double quotes and two backslash characters are included in the string. ■

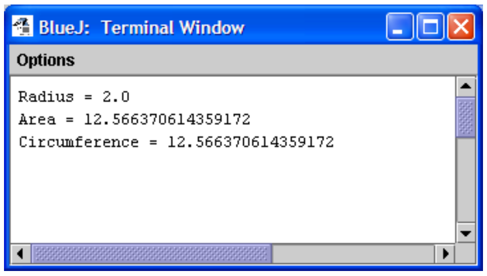
Using `System.out` in BlueJ

Using `System.out` or **BeanShell**'s `show` and `print` functions is the only way to test classes outside BlueJ but we don't normally need it in BlueJ. However BlueJ has a terminal window and if you use `System.out` it will automatically appear showing the output. We will find this useful later. For now, here is a small example.

■ EXAMPLE 4.14 **(Using `System.out` in BlueJ)** Add the method

```
public void display()
{
   System.out.println("Radius = " + radius);
   System.out.println("Area = " + area);
   System.out.println("Circumference = " + circumference);
}
```

to the `CircleCalculator` class (page 63). Now create an object for a radius of 3, choose its display method and you will see the terminal window shown in Figure 4.2. ■



```
Radius = 2.0
Area = 12.566370614359172
Circumference = 12.566370614359172
```

Figure 4.2: BlueJ terminal window

The `toString` method

The `toString` method is a special method in Java with the prototype

```
    public String toString()
```

The purpose of this method is to return a string representation of an object of the class.

An interesting property of this method is that if you use the name of an object in a string expression the object will be replaced by the result of calling its `toString` method. If the class does not contain a `toString` method a default one will be used.

■ EXAMPLE 4.15 **(Using `toString`)** The BeanShell statements

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = new CircleCalculator(3.0);
bsh % String rep = "toString gives " + circle;
bsh % print(rep);
toString gives CircleCalculator@4d1d41
bsh % rep = "toString gives " + circle.toString();
bsh % print(rep);
toString gives CircleCalculator@4d1d41
bsh % print(circle);
CircleCalculator@4d1d41
```

show what the default string representation looks like for a `CircleCalculator` object. It is not very useful. Notice that using the name `circle` is equivalent to using `circle.toString()` so the `toString` method essentially defines how an object can be converted to a string. The last statement shows that if the name of an object is used as an argument to `print` or `System.out.print` in Java then the object will be converted to a string value using the `toString` method. ■

Defining our own `toString` method

The default `toString` method is not very useful but we can redefine it in any of our classes and our version will be called.

■ EXAMPLE 4.16 **(Adding `toString` to `CircleCalculator`)** Add the method

```
public String toString()
{
   return "CircleCalculator[radius=" + radius + ", area=" +
      area + ", circumference=" + circumference + "]";
}
```

to the `CircleCalculator` class (page 63). Now create an object for a radius of 3, choose its `toString` method and you will see the result in a "Method Result" box. ■

■ EXAMPLE 4.17 **(Trying it with BeanShell)** Since we modified the class make sure you start a new version of BeanShell before trying the statements

```
bsh % addClassPath("c:/book-projects/chapter3");
bsh % CircleCalculator circle = new CircleCalculator(3.0);
bsh % print(circle);
CircleCalculator[radius=3.0, area=28.274333882308138,
circumference=18.84955592153876]
```

which show our string representation for a radius of 3. ■

The `toString` method is sometimes useful for finding logical errors in your classes. If you want to see the state of an object at a given place in the execution of a class you can insert a statement of the form

```
System.out.println(obj);
```

where `obj` is the name of the object.

4.2.5 Formatting numbers and strings (Java 5)

In Java 5 (Java 1.5) it is possible to format items (numbers and strings and other objects) according to the specifications provided in a **format string**. This gives precise control over how many columns are used for each item (field width), whether an item is left or right justified within its field width, and how many digits are displayed after the decimal point in the case of floating point numbers.

The String class contains a static method with prototype

```
public static String format(String f, Object... args)
```

Here `f` is the format string and `Object... args` represents the values to be formatted.

There is also a `printf` method that can be used with a format string that has the prototype

```
public void printf(String f, Object... args)
```

These methods are available in the latest version of BlueJ but not in BeanShell.

■ EXAMPLE 4.18 **(Format codes)** Here are some useful format codes (each begins with `%`).

`%5d`	format an integer right justified in field of width 5
`%-5d`	format an integer left justified in field of width 5
`%-20s`	format a string left justified in a field of width 20
`%15.5f`	format a floating point number right justified in a field of width 15 using fixed format rounded to 5 digits after the decimal point
`%.5f`	format a floating point number in a field that just fits using fixed format rounded to 5 digits after the decimal point
`%20.8e`	format a floating point number right justified in a field of width 20 using exponential (scientific) format rounded to 8 digits after the decimal point

There are many other types of codes. A complete list can be found in the Java documentation for the `Formatter` class. ■

■ EXAMPLE 4.19 **(Formatted strings)** The statements

```
int i = 3;
double pi = Math.PI;
String end = "End";
String f = String.format("answer: %5d%15.5f%10s", i, pi, end);
System.out.println(f);
```

produces the output

```
answer:     3        3.14159       End
```

consisting of the literal string `"answer: "` followed by the integer 3 right justified in a field of width 5 followed by the value of π right justified in a field of width 15 and rounded to 5 digits after the decimal point, and followed by the value of the string `end` right justified in a field of width 10.

The `printf` method can also be used to specify the format string and print it. The statements

```
int i = 3;
double pi = Math.PI;
String end = "End";
System.out.printf("answer: %5d%15.5f%10s\n", i, pi, end);
```

produce the same output if we add the newline character `\n` to the end of the format string. ■

■ EXAMPLE 4.20 **(Using `System.out.printf` in BlueJ)** Repeat Example 4.14 using the following method.

```
public void display()
{
   System.out.printf("Radius = %.5f\n", radius);
   System.out.printf("Area = %.5f\n", area);
   System.out.printf("Circumference = %.5f\n", circumference);
}
```

to display the values rounded to 5 digits. ■

4.3 Example classes that use the `String` class

We now write some simple classes that use the `String` class. We also illustrate a three step "design, implement, test" process for writing classes.

1. Begin with an English description of the class.
2. Design the class by deciding what methods it should have. This is called writing the class **specification** or the **public interface**. If the class is designed to be used primarily by other classes (the `String` class is an example) then check the convenience of your design by writing some typical statements that use the class.
3. Write the complete class by providing the implementation. This involves choosing any instance data fields, and providing bodies for all methods and even providing `private` methods, if necessary, to aid in the implementation.
4. Test the class by itself using BlueJ or BeanShell, or both. Even if the class is designed to be used by other classes it should be tested by itself before being used in a larger system of classes. This is called **unit testing**.

4.3.1 `BankAccount` class (first version)

A description of this class is

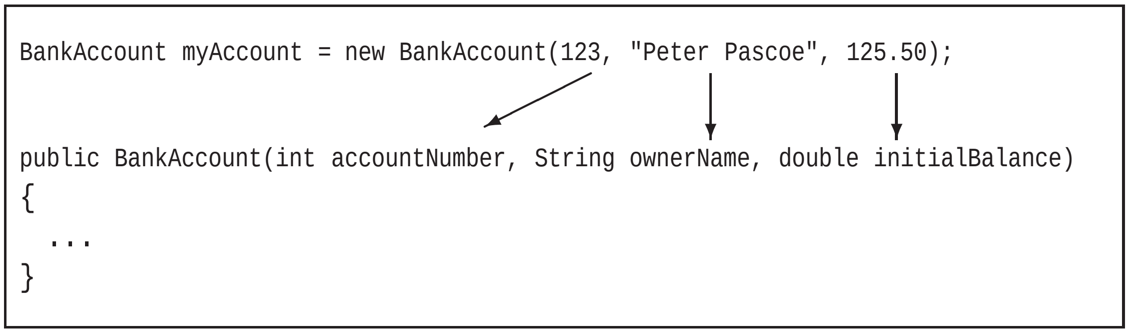
> *"A `BankAccount` object should represent a bank account using an account number, an owner name, and a current balance. There should be a constructor for creating a bank account given these values. There should be methods to withdraw or deposit a given amount and the usual "get methods" for returning the account number, owner name, and balance."*

This is a **mutable** class since the withdraw and deposit methods change the balance in the account. We use this class many times throughout the book to illustrate important concepts.

Designing the class

The English description directly gives the following specification or public interface.

```
public class BankAccount
{
   // put instance data field declarations here
   public BankAccount(int accountNumber, String ownerName,
      double initialBalance) {...}
   public void deposit(double amount) {...}
   public void withdraw(double amount) {...}
```



```
BankAccount myAccount = new BankAccount(123, "Peter Pascoe", 125.50);


public BankAccount(int accountNumber, String ownerName, double initialBalance)
{
  ...
}
```

Figure 4.3: Matching actual and formal constructor arguments

```
   public int getNumber() {...}
   public String getOwner() {...}
   public double getBalance() {...}
}
```

We have used the notation `{...}` to indicate that the method bodies are part of the implementation step, not the public interface step. Also we have not shown the javadoc comments but they should also be included in this step. We have not indicated the instance data fields yet.

For this simple class it is easy to check our design. For example, to create an account, withdraw $100, and show the current balance we could use the statements

```
BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
account.withdraw(100);
System.out.println("The current balance is " + account.getBalance());
```

The correspondence between the formal arguments in the constructor prototype

```
public BankAccount(int accountNumber, String ownerName,
   double initialBalance)
```

and the actual arguments in the constructor call expression

```
new BankAccount(123, "Peter Pascoe", 125.50)
```

is shown in Figure 4.3.

Implementing the class

We need to declare three private instance data fields for the account number, account owner name, and current balance:

```
private int number;
private String name;
private double balance;
```

There is one set of these variables for each bank account object and the purpose of the constructor is to initialize these fields. Therefore, we can write the constructor as

```
public BankAccount(int accountNumber, String ownerName, double initialBalance)
{
   number = accountNumber;
   name = ownerName;
   balance = initialBalance;
}
```

The deposit method needs to add the amount specified by the formal argument to the value of the instance data field for the balance and the withdraw method needs to subtract this amount. Therefore, we can write the `deposit` method as

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

and the `withdraw` method

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

The “get methods” are easy to write since they just return the value of one of the instance data fields. For example,

```
public String getName()
{
   return name;
}
```

We can also add a `toString` method. Here is the completed class declaration complete with javadoc comments.

Class `BankAccount`

book-projects/chapter4/bank_account

```
package chapter4.bank_account; // remove this line if you’re not using packages
/**
 * First version of the BankAccount class. No error checking
 * is performed.
 * <p>
 * There is a better version in library.BankAccount.
 * <p>
 * Each object from this class encapsulates the account number,
 * owner name, and current balance of a bank account.
 */
public class BankAccount
{
   private int number;
   private String name;
   private double balance;

   /**
    * Construct a bank account with given account number, owner name
    * and initial balance.
    * @param accountNumber the account number.
    * @param ownerName the owner name.
    * @param initialBalance the initial balance.
   */
   public BankAccount(int accountNumber, String ownerName,
      double initialBalance)
   {
      number = accountNumber;
      name = ownerName;
      balance = initialBalance;
   }

   /**
    * Deposit the given amount of money in the account.
    * @param amount the amount to deposit (no error checking).
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /**
    * Withdraw the given amount of money from the account.
    * @param amount the amount to withdraw (no error checking).
    */
   public void withdraw(double amount)
   {
      balance = balance - amount;
   }

   /**
    * Return the account number.
    * @return the account number.
    */
   public int getNumber()
   {
      return number;
   }

   /**
    * Return the owner name.
    * @return the owner name.
    */
   public String getName()
   {
      return name;
   }

   /**
    * Return the account balance.
    * @return the account balance.
    */
   public double getBalance()
   {
      return balance;
   }

   /**
    * Return a string representation of a bank account.
    * @return a string representation of a bank account.
    */
   public String toString()
   {
       return "BankAccount[number=" + number + ", name="
          + name + ", balance=" + balance + "]";
   }
}
```

Testing the class

This is only a preliminary version of this class. Later when we have introduced conditional statements we can make a more robust version of this class that checks, for example, if the amount specified in the `withdraw` method would not result in an overdrawn account.

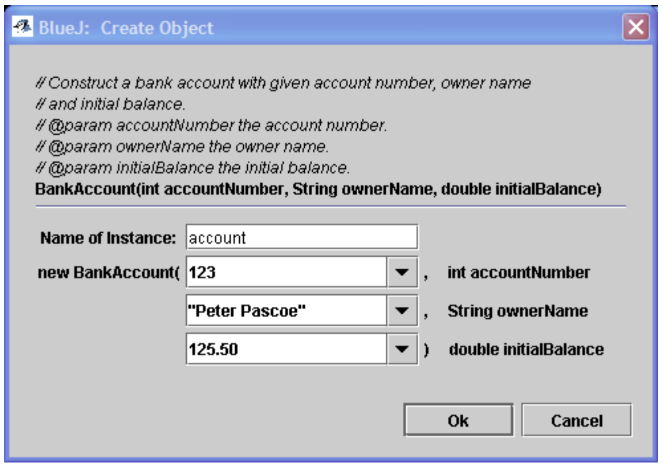
This class is easy to test in BlueJ. For example, construct an object called `circle` as shown in Figure 4.4(a) and choose some of its methods from the method menu shown in Figure 4.4(b).

You can also test this class using either the BeanShell workspace or the BeanShell editor.

■ EXAMPLE 4.21 **(Testing `BankAccount` with BeanShell)** The following statements

```
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
bsh % account.withdraw(100);
bsh % print(account.getBalance());
25.5
bsh % account.deposit(100);
bsh % print(account.getBalance());
125.5
bsh % print(account);
BankAccount[number=123, name=Peter Pascoe, balance=125.5]
```

show how to create an account, withdraw and deposit $100, and use the `toString` method to display the account. ■

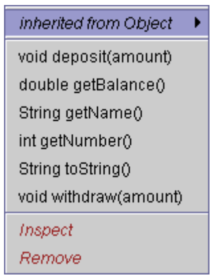


BlueJ: Create Object

```
// Construct a bank account with given account number, owner name
// and initial balance.
// @param accountNumber the account number.
// @param ownerName the owner name.
// @param initialBalance the initial balance.
BankAccount(int accountNumber, String ownerName, double initialBalance)
```

Name of Instance:	account	
new BankAccount(	123 ,	int accountNumber
	"Peter Pascoe" ,	String ownerName
	125.50)	double initialBalance

(a)



(b)

Figure 4.4: (a) Constructing a `BankAccount` object, (b) the object menu

Testing is important even in a simple class like this. While writing this book the author used the `toString` method

```
public String toString()
{
    return "BankAccount[number=" + name + ", name="
       + name + ", balance=" + balance + "]";
}
```

Can you find the logical error? The error was noticed when BeanShell produced the result

```
bsh % print(account);
BankAccount[number=Peter Pascoe, name=Peter Pascoe, balance=125.5]
```

4.3.2 `InitialsMaker` class

A description of this class is

> *"An `InitialsMaker` object uses the first and last name of a person to produce the initials. For example, if the name is Henry James then the initials are HJ."*

Designing the class

We need a constructor that has two string arguments for the first and last names, a method to return the initials as a string and we will also include a `toString` method.

This gives the public class interface

```
public class InitialsMaker
{
   // instance data fields go here
   public InitialsMaker(String firstName, String lastName) {...}
   public String getInitials() {...}
   public String toString() {...}
}
```

We choose to make this an immutable class so no "set methods" are provided.

Implementing the class

The class is easily implemented. We choose one instance data field of type String for the initials:

```
public String initials
```

We have made a decision here not to include the first and last names as data fields, so we do not include "get" methods for them. An alternate design would be to provide these data fields and their associated "get methods".

The constructor can use the `substring` method to extract the first letter of each name. Then these letters can be concatenated together to get a two character string. Finally, this string can be converted to upper case:

```
public InitialsMaker(String firstName, String lastName)
{
   initials = firstName.substring(0,1) + lastName.substring(0,1);
   initials = initials.toUpperCase();
}
```

The complete class is given by

Class `InitialsMaker`

book-projects/chapter4/strings

```
package chapter4.strings; // remove this line if you're not using packages
 /**
  * An object of this class takes a person's first and last names
  * and extracts the first letter of each name
  * to makes a two character initial string converted to upper case.
  */
public class InitialsMaker
{
   private String initials; // string containing two initials

   /**
    * Construct an object for the given names
    * @param firstName the first name
    * @param lastName the last name
    */
   public InitialsMaker(String firstName, String lastName)
   {
      initials = firstName.substring(0,1) + lastName.substring(0,1);
      initials = initials.toUpperCase();
   }

   /**
    * Return the initial string
    * @return return the initial string
    */
   public String getInitials()
   {
      return initials;
   }

   public String toString()
   {
      return "InitialsMaker[initials=" + initials + "]";
   }
}
```

Testing the class

The class is easily tested in both BlueJ and BeanShell. There are two kinds of tests: (a) are the first letters of each name being extracted properly, and (b) are the initials being converted to upper case properly. For example, names like ”Fred Duncan” can test (a) but not (b), where it is necessary to try names such as “henry james”, “Henry james”, and ”henry James”.

■ EXAMPLE 4.22 **(Testing `InitialsMaker` with BeanShell)** The following statements

```
bsh % addClassPath("c:/book-projects/chapter4/strings");
bsh % InitialsMaker maker = new InitialsMaker("harry", "james");
bsh % print(maker.getInitials());
HJ
bsh % print(maker);
InitialsMaker[initials=HJ]
```

show how to create an object and perform one of the tests on it. ■

4.3.3 `PasswordGenerator` class

A description of this class is

> *“A `PasswordGenerator` object generates random 7 character passwords. The first four characters should be lower case letters and the last three characters should be digits 0 to 9.”*

Designing the class

Since no input is required we only need the constructor with no arguments. Also we need a method called `next` which will return the next random password as a string each time it is called. This gives the simple class interface

```
public class PasswordGenerator
{
   // instance data fields go here
   public PasswordGenerator() {...}
   public String next() {...}
}
```

Implementing the class

Before implementing this class we need to find out how to generate random numbers. Either we have to do it ourselves or we find out if Java can do it. Fortunately, Java can do it so we will follow the “do not reinvent the wheel policy”.

In Example 2.27 (page 29) we showed how to use the `Math.random` method in the `Math` class. This method can generate random `double` numbers in the range $0 \le r < 1$ which were converted to integers in the range $1 \le i \le 10$. We could use this approach but there is also a class called `Random` whose objects can generate random integers directly. It has two constructors with prototypes

```
public Random()
public Random(long seed)
```

The first constructor is used to generate a sequence of random numbers that depends on the current time in milliseconds. In other words the same sequence will not be repeated. The second constructor generates sequences that use a “seed”. Each value of the seed gives a repeatable sequence. Of course if you are using a random number generator in a game you will not want to use this version since every time you run the game the same sequence will be generated.

This suggests that we modify our design to use two constructors. The no-arg constructor can use the current time to generate a seed and the other one can use a specified seed. This gives the modified class design

```
public class PasswordGenerator
{
   // instance data fields go here
   public PasswordGenerator() {...}
   public PasswordGenerator(long seed) {...}
   public String next() {...}
}
```

This is our first example of a class that has more than one constructor. This is permissible and quite common as long as the constructors can be distinguished by their argument types (signature).

The `Random` class has several methods but the one we are interested in has the prototype

```
public int nextInt(int n)
```

which generates a random integer i in the range $0 \le i \le n-1$. We now have enough information to write the following partial implementation of the class

```
public class PasswordGenerator
{
   private Random random;

   // any other instance data fields go here

   public PasswordGenerator()
   {
      random = new Random();
   }
   public PasswordGenerator(long seed)
   {
      random = new Random(seed);
   }
   public String next() {...}
}
```

Here we have an object reference `random` being used as an instance data field. It refers to a `Random` object in the same way that `name` in the `BankAccount` class (page 106) refers to a `String` object. Thus, each constructor needs to create an object and assign its reference to `random`.

To implement the `next` method we need to first generate random characters in the range `'a'` to `'z'`. This can be done by generating random integers in the range 0 to 25 using the method call expression `random.nextInt(26)` and using the result as an index into the string

```
   String LETTERS = "abcdefghijklmnopqrstuvwxyz";
```

to generate a random letter. Starting with an empty string we have

```
   int index;
   String password = "";
   index = random.nextInt(26);
   password = password + LETTERS.substring(index, index+1);
```

Repeating the last two statements three more times gives us a string of four random letters.

To generate random digit characters we can simply use `random.nextInt(10)` and convert the result to a digit character by concatenation onto the letter string.

The `LETTERS` string is an example of a constant string and `PasswordGenerator` objects do not need their own copies of this string. One copy for all objects suffices and this can be indicated using the `static` modifier. Static data fields are **not** instance data fields:

```
   private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
```

This gives the following complete class declaration:

Class `PasswordGenerator`

book-projects/chapter4/strings

```
package chapter4.strings; // remove this line if you're not using packages
import java.util.Random;
/**
```

```
 * An object of this class knows how to generate
 * random 7 character password of the form LLLLDDD where L
 * is a lower case letter and D is a digit.
 */
public class PasswordGenerator
{
   private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
   private Random random;

   /**
    * Construct an default generator whose sequence is based on the
    * current time in milliseconds.
    */
   public PasswordGenerator()
   {
      random = new Random();
   }

   /**
    * Construct a generator that is repeatable. If the same seed
    * is used again then the same sequence is generated.
    * @param seed a seed to start the random number generator.
    */
   public PasswordGenerator(long seed)
   {
      random = new Random(seed);
   }

   /**
    * Return a generated password. Another password is generated each
    * time this method is called.
    * @return another password
    */
   public String next()
   {
      int index; // index into a string (0,1,2,...)

      String password = "";

      index = random.nextInt(26); // 0 to 25 inclusive
      password = password + LETTERS.substring(index, index + 1);
      index = random.nextInt(26);
      password = password + LETTERS.substring(index, index + 1);
      index = random.nextInt(26);
      password = password + LETTERS.substring(index, index + 1);
      index = random.nextInt(26);
      password = password + LETTERS.substring(index, index + 1);

      index = random.nextInt(10);
      password = password + index;
      index = random.nextInt(10);
      password = password + index;
      index = random.nextInt(10);
      password = password + index;

      return password;
   }
}
```

The import statement The very first line of this class is new. The `Random` class is not a standard class like `Math`, `System`, and `String`. These classes are in a Java **package** (collection of related classes) called `java.lang` that is automatically imported into any class that needs them. Thus, for example, the **fully qualified name** of the `String` class is `java.lang.String`.

For Java classes in other packages such as `java.util` it is necessary to explicitly import them. For `Random` this is done using

```
import java.util.Random;
```

before the class declaration.

It is not essential to use the `import` statement. Its purpose is simply to allow you to use the short names instead of the fully qualified ones. Thus, if `Random` is not imported then we must use its full name `java.util.Random` everywhere. For example, as a private data field we would need to write

```
private java.util.Random random;
```

and in the constructors we would need to write

```
random = java.util.Random();
random = java.util.Random(seed);
```

Testing the class

This class is easily tested either with BeanShell or with BlueJ. The following example shows how to test the class in BeanShell.

■ EXAMPLE 4.23 **(Testing `PasswordGenerator` with BeanShell)** The following statements

```
bsh % addClassPath("c:/book-projects/chapter4/strings");
bsh % PasswordGenerator gen = new PasswordGenerator();
bsh % print(gen.next());
avfi637
bsh % print(gen.next());
iqde665
bsh % gen = new PasswordGenerator(); // make a new one
bsh % print(gen.next());
zuwe456
bsh % gen = new PasswordGenerator(123);
bsh % print(gen.next());
```

```
eomt574
bsh % gen = new PasswordGenerator(123);
bsh % print(gen.next());
eomt574
bsh %
```

show how to create objects and test them. Four `PasswordGenerator` objects are created here and references to them are assigned to the reference variable `gen`.

The first two are based on the current time so they produce different values the first time their `next` method is called but the last two use the same seed so the first time their `next` method is called the same password `eomt574` is produced. ■

With BlueJ try constructing objects of each type (current time or seed) and try the `next` method on the object menu several times. Another approach is to add the following test method to the `PasswordGenerator` class:

```
public void test()
{
   System.out.println(next());
}
```

Each time we select this method from the object menu a random password is appended to the BlueJ terminal window.

The "`this`" object

There is an important idea in the above test method. Notice that `next()` is used without applying it on an object, whereas in BeanShell Example 4.23 we used `gen.next()` since we already had the variable `gen`. The BeanShell statements were not part of the `PasswordGenerator` class. They were simply statements that used the class.

However, the `next()` method belongs to the `PasswordGenerator` class itself and we do not have an explicit object name like `gen` to refer to. In a situation like this we use what is called the `this` object. Using `next()` is equivalent to using `this.next()`.

We have seen `this` before in the `QuadraticRootFinder` class (page 67). There it was used to refer to an instance data field when an argument had the same name. Now we are using it to refer to "this" object. The compiler will also accept

```
System.out.println(this.next());
```

where `this` is explicit but it is not necessary.

Whenever you see an instance method call expression in a class that is not prefixed with an object name, `this` is the implied object. In the `doCalculations` method in `QuadraticRootFinder` `this` was implied and we could have used

```
this.doCalculations()
```

4.4 Association and aggregation

Our Java programs normally consist of interacting objects from several classes. These classes are either built-in classes such as `String` that come with the Java SDK (software development kit), classes obtained from someone else, such as your course instructor or friends, or classes we write ourselves such as `CircleCalculator` or `PasswordGenerator`. We will also see that complex classes are commonly designed in terms of simpler ones. The terms **association** and **aggregation** are often used to describe how classes can relate to each other.

4.4.1 Association

Some classes are not related to any other classes. For example, the classes in Chapter 3 do not require any other class in order be compiled and work properly.

On the other hand, the `PasswordGenerator` class (page 113) depends on both the `String` and `Random` classes. If either of these classes were not available then the class could not be compiled. We say that the `PasswordGenerator` class is associated with, or uses, these classes. This relationship is not symmetric so we do not say that the `String` class is associated with the `PasswordGenerator` class. Classes, such as the `String` class, are designed to be used by other classes.

Thus, class `A` is associated with class `B` if `A` uses `B`. This can occur in several ways:

1. An object of `B` is used as a local variable in a constructor or method in `A`.
2. An object of `B` is used as a method or constructor argument in `A`.
3. An object of `B` is used as a return value of a method in `A`.
4. An object of `B` is used as an instance data field in `A`.

When we use the word object here we really mean an object reference. We will see many examples of these four kinds of relationships.

4.4.2 Aggregation

Case (4) above is an important special case of association called aggregation. This is how we make complex objects out of simpler ones and is often used in a “bottom up” approach to object-oriented design. First we design, implement, and test the simplest classes and then we use them as instance data fields of more complex classes, and so on.

We have already seen some simple examples of aggregation in the classes in this Chapter that use `String` objects as instance data fields and the `PasswordGenerator` class that uses a `Random` object.

4.4.3 `TriangleCalculatorTester` class

As an example of association in which one class uses a local variable of another class consider the following small class that could be used to test the `TriangleCalculator` class.

Class `TriangleCalculatorTester`

book-projects/chapter4/tester

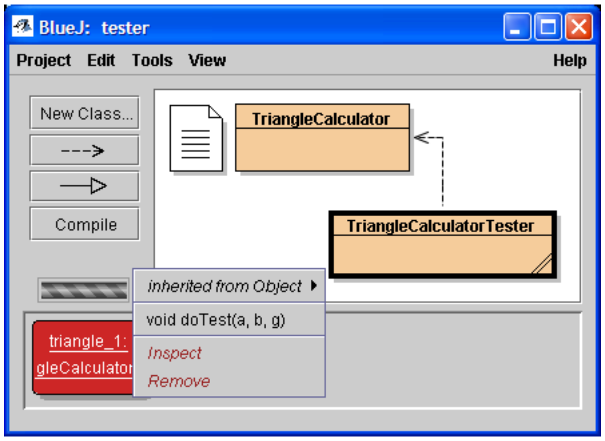
```
package chapter4.tester; // remove this line if you’re not using packages
import chapter3.TriangleCalculator; // remove this line if you’re not using packages
/**
 * A short class to show how to test the TriangleCalculator
 * class from Chapter3 using System.out.println
 */
public class TriangleCalculatorTester
{
   public TriangleCalculatorTester()
   {
   }

   /**
    * Test the TriangleCalculator class
    * @param a side length
    * @param b another side length
    * @param g contained angle in degrees
    */
   public void doTest(double a, double b, double g)
   {
      TriangleCalculator tri = new TriangleCalculator(a,b,g);
      System.out.println("Sides:  " + tri.getA() + ", " + tri.getB()
         + ", " + tri.getC());
      System.out.println("Angles: " + tri.getAlpha() + ", " + tri.getBeta()
         + ", " + tri.getGamma());
      System.out.println("Angle sum is " + tri.checkAngleSum());
   }
}
```

In this class the constructor has nothing to do so its body is empty. The `doTest` method does all the work by creating a `TriangleCalculator` object as a local variable and displaying the results in the terminal window.

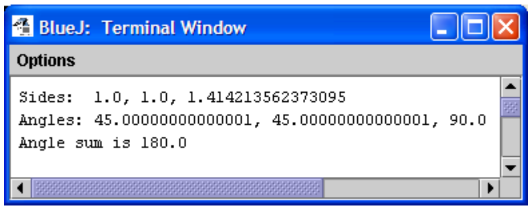
This class is in BlueJ project `book-projects/chapter4/tester`, as shown in Figure 4.5(a). We have also included with the `TriangleCalculator` class from Chapter 3 (page 64) which was in the `book-projects/chapter3` project. The dashed line from the `TriangleCalculatorTester` class to the `TriangleCalculator` class indicates the association or “uses” relation. The results of constructing an object with $a = 1$, $b = 1$ and $c = 90$ degrees is shown in the terminal window in Figure 4.5(b).

4.4.4 `Point` class

As an another example of association and the idea of building objects from simpler objects we first consider a `Point` class for geometrical points (x, y). Then we will use it to write a `Circle` class.



(a)



```
Sides:  1.0, 1.0, 1.414213562373095
Angles: 45.00000000000001, 45.00000000000001, 90.0
Angle sum is 180.0
```

(b)

Figure 4.5: (a) Association in BlueJ using dashed line, (b) output for a test case.

Designing the class

The `Point` class has instance data fields of type `double` for the *x* and *y* coordinates of a point, constructors, ''get methods" for the coordinates, and we include a `toString` method. We also assume that this class is immutable. Therefore, the public interface is

```
public class Point
{
   double x, y;
   public Point() {...}
   public Point(double x, double y) {...}
   public double getX() {...}
   public double getY() {...}
   public String toString() {...}
}
```

For the constructor with no arguments we choose to construct the point at the origin with coordinates $(0,0)$. Many other methods could be included (see end of Chapter exercises).

Implementing the class

Here is the complete implementation of the class.

Class `Point`

book-projects/chapter4/geometry

```
package chapter4.geometry; // remove this line if you're not using packages
/**
 * A class representing immutable geometrical points (x,y)
 * in the plane.
 */
public class Point
{
   private double x;
   private double y;

   /**
    * Construct a point from its coordinates.
    * @param x the x coordinate of the point
    * @param y the y coordinate of the point
    */
   public Point(double x, double y)
   {
     this.x = x;
     this.y = y;
   }

   /**
    * Construct the default point (0,0).
    */
   public Point()
   {
      x = 0.0;
      y = 0.0;
   }

   /**
    * Return the x coordinate of this point.
    * @return the x coordinate of this point
    */
   public double getX()
   {
      return x;
   }

   /**
    * Return the y coordinate of this point.
    * @return the y coordinate of this point
    */
   public double getY()
   {
      return y;
   }

   /**
    * Return a string representation of a Point.
    * @return a string representation of a Point
    */
   public String toString()
   {
      return "Point[" + x + ", " + y + "]";
   }
}
```

Testing the class

This class is easily tested in BlueJ. With BeanShell we have to be careful as the following example shows.

■ EXAMPLE 4.24 **(Testing `Point` with BeanShell)** The following statements

```
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point origin = new Point();
bsh % Point p = new Point(1,2);
bsh % print(origin);
Point[0.0, 0.0]
bsh % print(p);
Point[1.0, 2.0]
bsh % print(p.getX());
1.0
bsh % print(p.getY());
2.0
```

show how to create some `Point` objects and test them. It is necessary here to use the `import` statement.[1] ■

4.4.5 `Circle` class

The `Circle` class describes a circle in terms of its radius and the *x* and *y* coordinates of its center.

Designing the class

Our first attempt at an interface for this class might be

```
public class Circle
{
   private double x, y, radius;
   public Circle() {...}
   public Circle(double x, double y, doubler r) {...}
   public double getX() {...}
   public double getY() {...}
   public double getRadius() {...}
   public String toString() {...}
}
```

[1] There is already a class in package `java.awt` called `Point` and BlueJ will use it by default unless we tell it to use our version. Therefore we need the `import` statement.

Here we have two constructors. The no-arg constructor is for the unit circle, with radius 1 and center $(0,0)$. The next one uses the *x* and *y* coordinates of the center and the radius to define the circle.

In this design we are not associating the `Circle` class with the `Point` class so there would be no association between the two classes. However, since we already have the `Point` class it would be better to use aggregation and use the instance data fields and interface given by

```
public class Circle
{
   private Point center;
   private double radius;
   public Circle() {...}
   public Circle(double x, double y, doubler r) {...}
   public Circle(Point c, double r) {...}
   public Point getCenter() {...}
   public double getRadius() {...}
   public String toString() {...}
}
```

Now we choose 3 constructors. The first two have the same prototypes as in the previous design but now the third one allows a `Point` object to be used as a reference argument. Now we have a `getCenter` method that returns a reference to the center of the circle as a `Point` object. We could have also included the `getX` and `getY` method but they can be obtained using the corresponding methods of the `Point` class applied to the `Point` object returned by `getCenter`.

Implementing the class

Here is the completed `Circle` class.

Class `Circle`

book-projects/chapter4/geometry

```
package chapter4.geometry; // remove this line if you’re not using packages
/**
 * A class representing immutable geometrical circles.
 * Each circle is described by its center (a Point object)
 * and its radius (a double number).
 */
public class Circle
{
   private Point center;
   private double radius;

   /**
    * Construct circle with given center point and radius.
    * @param p the center of the circle
    * @param r the radius of the circle
    */
```

```
public Circle(Point p, double r)
{
   center = p;
   radius = r;
}

/**
 * Construct circle with given center coordinates and radius.
 * @param x the x coordinate of the circle center
 * @param y the y coordinate of the circle center
 * @param r the radius of the circle
 */
public Circle(double x, double y, double r)
{
   center = new Point(x,y);
   radius = r;
}

/**
 * Construct a default circle: a unit circle with center (0,0)
 * and radius 1.
 */
public Circle()
{
   center = new Point();
   radius = 1;
}

/**
 * Return radius of circle.
 * @return radius of circle
 */
public double getRadius()
{
   return radius;
}

/**
 * Return center of circle.
 * @return center of circle
 */
public Point getCenter()
{
   return center;
}

/**
 * Return a string representation of a Circle.
 * @return a string representation of a Circle
 */
public String toString()
{
      return "Circle[" + center + ", " + radius + "]";
   }
}
```

Recall that when an object name is used in a string it is replaced by the result of calling its `toString` method. We have used this to include the `toString` method of the `Point` class in the `toString` method of the `Circle` class.

Testing the class with BeanShell

The following example shows how to test the `circle` class in BeanShell.

■ EXAMPLE 4.25 **(Testing `Circle` with BeanShell)** Try the following statements in BeanShell

```
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point center = new Point(3,4);
bsh % Circle c1 = new Circle();
bsh % Circle c2 = new Circle(center, 5);
bsh % Circle c3 = new Circle(3,4,5);
bsh % print(c1);
Circle[Point[0.0, 0.0], 1.0]
bsh % print(c2);
Circle[Point[3.0, 4.0], 5.0]
bsh % print(c3);
Circle[Point[3.0, 4.0], 5.0]
bsh % double radius = c2.getRadius();
bsh % double x = c2.getCenter().getX();
bsh % double y = c2.getCenter().getY();
bsh % print(radius);
5.0
bsh % print(x);
3.0
bsh % print(y);
4.0
```

Here we construct a default circle `c1` and a circles `c2` and `c2` with center at $(3,4)$ and radius 5 in two ways using the second and third constructors. Then we use the `toString` method to display the results and show how the "get" methods are used. This also shows that we did not need to provide "get" methods in the `Circle` class for the `x` and `y` coordinates of the center since, for example, the `x` coordinate can be obtained using the method call expression `c2.getCenter().getX()` which uses two methods in one expression. First we get the center as `c2.getCenter()`. Since this is an object of type `Point` we can then invoke its `getX()` method all in one statement, and similarly for the `y` coordinate. When aggregation is involved it is common to see several method invocations strung together like this. In this case the separate statements

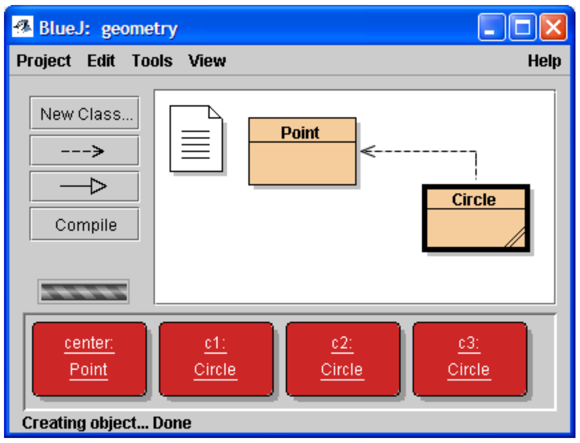


Figure 4.6: Aggregation with the `Point` and `Circle` classes.

```
Point c = c2.getCenter();
double x = c.getX();
double y = c.getY();
```

could also be used. ■

Testing the class with BlueJ

Because the `Circle` class uses aggregation it is interesting to test it in BlueJ. To do this construct four objects as follows so that you obtain the project shown in Figure 4.6.

1. Right click on the `Point` rectangle and select menu item `new Point(x,y)` to construct a point (3,4) with name `center`.

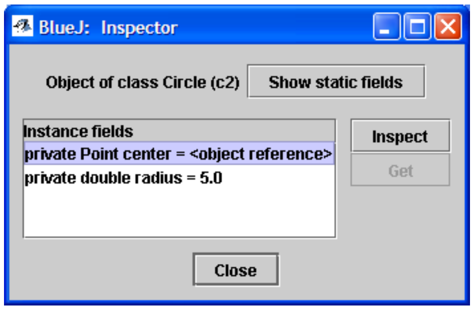
2. Right click on the `Circle` rectangle and select menu item `new Circle()` to construct a default circle with name `c1`.

3. Right click on the `Circle` rectangle again and select menu item `new Circle(p,r)` to construct a circle with name `c2`. Choose `center` as the center point. You can do it in the "Create Object' dialog box by typing its name in the box, or just click in the text field for the center point and then click on the `center` object and its name should appear in the text field. Finally, choose 5 for the radius.

4. As above but choose menu item `new Center(x,y,r)` and enter 3, 4, and 5 for the values `x`, `y`, and `r`.

You should now have the four objects on the workbench as shown in Figure 4.6. The fact that the `Circle` class "uses" the `Point` class is indicated by the dashed arrow.

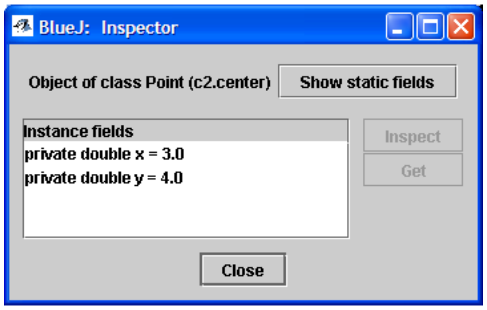
Now you can right click on any of the objects and choose a method or the "inspect" menu choice to test the class. To see aggregation and object references in action try the following



Object of class Circle (c2)

Instance fields
private Point center = <object reference>
private double radius = 5.0

(a)

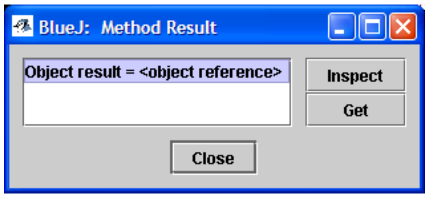


Object of class Point (c2.center)

Instance fields
private double x = 3.0
private double y = 4.0

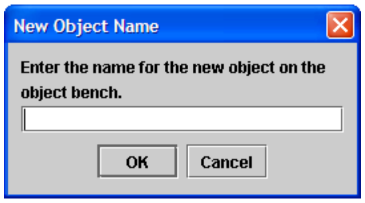
(b)

Figure 4.7: (a) Choosing “Inspect” from the `c2` menu, (b) choosing “Inspect” for the resulting object reference.



Object result = <object reference>

(a)



Enter the name for the new object on the object bench.

(b)

Figure 4.8: (a) Choosing `getCenter` from `c2` menu, (b) result of using ”get” from the “Method Result” box.

1. Right click on object `c2` and choose the “Inspect” menu choice to get the “Inspector” box shown in Figure 4.7(a). Instead of showing the `Point` object fields we only see `<object reference>`. This is BlueJ’s way of telling us that `center` is a reference to the object, not the object itself.

2. Now click on the this reference and click the “Inspect” button and you will actually see the `x` and `y` fields for the `center` object.

The `Circle` class uses aggregation by having a `Point` object as an instance data field but it also uses association in another way because the `getCenter` method returns a reference to the `Point` object that is the center of the circle. BlueJ lets you put such objects on the work bench as follows.

1. From the `c2` object menu select the `getCenter` method to get the “Method Result” box shown in Figure 4.8(a).

2. Click on the object reference and click the “Get” button to get the box shown in Figure 4.8(b) asking for the name of the object

3. Choose a name and the object will appear on the object bench.

We can also write classes, similar to `TriangleCalculatorTester` (page 118), to test the `Point` and `Circle` classes. Here is a simple example:

Class `CircleTester`

book-projects/chapter4/geometry

```
package chapter4.geometry; // remove this line if you’re not using packages
/**
 * A short class to show how to test the Circle and Point classes.
 */
public class CircleTester
{
   public CircleTester()
   {
   }

   /**
    * Test the Point and Circle classes.
    */
   public void doTest()
   {
      Point center = new Point(3,4);
      Circle c1 = new Circle();
      Circle c2 = new Circle(center, 5);
      Circle c3 = new Circle(3, 4, 5);
      System.out.println("c1 = " + c1);
      System.out.println("c2 = " + c2);
      System.out.println("c3 = " + c3);

      double radius = c2.getRadius();
      double x = c2.getCenter().getX();
      double y = c2.getCenter().getY();
      System.out.println("Radius = " + radius);
      System.out.println("Center x = " + x);
      System.out.println("Center y = " + y);
   }
}
```

4.5 Other library classes

We have used the `String` and `Random` library classes. There are many others that are useful and we illustrate a few of them here.

4.5.1 Dates and times

There are three classes that are useful for working with dates and formating them.

`Date` class

The `Date` class in package `java.util` has two constructors and two useful methods, among others, with prototypes

```
public Date()
public Date(long date)
public long getTime()
public void setTime(long date)
public String toString()
```

The first constructor creates a `Date` object for "right now". In this class each date is represented by a `long` integer that is the number of milliseconds since January, 1, 1970, 00:00:00 GMT. For example, while I am typing this the date is 1055681816162. The `toString` method gives the more meaningful result "Sun Jun 15 08:56:56 EDT 2003".

■ EXAMPLE 4.26 **(The `Date` class using BeanShell)** Try the following statements in BeanShell

```
bsh % import java.util.Date;
bsh % Date now = new Date();
bsh % print(now);
Sun Jun 15 08:56:56 EDT 2003
bsh % long t = now.getTime();
bsh % print(t);
1055681816162
bsh % Date first = new Date(0L);
bsh % print(first);
Wed Dec 31 19:00:00 EST 1969
bsh % first.setTime(0L + 1000L * 60L * 60L * 24L);
bsh % print(first);
Thu Jan 01 19:00:00 EST 1970
bsh %
```

The `import` statement is necessary here. Also recall that a `long` integer literal needs an `L` suffix to distinguish it from an `int` literal. The `first` date corresponds to time 0, indicating that we are 5 hours behind GMT. Then 1 day is added to this time by adding to `0L` the number of milliseconds in a day. ■

`SimpleDateFormat` class

The `SimpleDateFormat` class in package `java.text` can be used to format dates in many different ways. Two of its constructors and a format method have prototypes

```
public SimpleDateFormat()
public SimpleDateFormat(String pattern)
public String format(Date d)
```

This class is associated with the `Date` class. The first constructor uses a default format specific to the user locale and the second constructor has an argument to specify the format. The `format` method formats a given `Date` object as a `String` using the specified format.

■ EXAMPLE 4.27 **(Using `SimpleDateFormat`)** Continuing the previous example try the following statements in BeanShell

```
bsh % import java.text.SimpleDateFormat;
bsh % SimpleDateFormat f1 = new SimpleDateFormat();
bsh % String n1 = f1.format(now);
bsh % print(n1);
6/15/03 8:56 AM

bsh % SimpleDateFormat f2 = new SimpleDateFormat("dd/MM/yyyy");
bsh % String n2 = f2.format(now);
bsh % print(n2);
15/06/2003

bsh % SimpleDateFormat f3 = new SimpleDateFormat("HH:mm:ss z");
bsh % String n3 = f3.format(now);
bsh % print(n3);
08:56:56 EDT
```

Again the `import` statement is necessary here. This example shows three of the many different date formats that are possible[2]. ■

`Calendar` class

The `Date` class does not deal with the many properties of dates such as the year, month, day of the month, day of the year, etc. The `Calendar` class in package `java.util`, has this functionality.

There are several different calendars in use around the world and Java can use any of them. In particular we are interested in the class called `GregorianCalendar`. Constructors for the `Calendar` class would be quite complex and would have to deal with the different conventions such as Sunday being the first day of the week in North American countries but Monday being the first day of the week in France.

Therefore we do not have constructors in the `Calendar` class. Instead a static `getInstance` method is provided to return the appropriate calendar object for your locale. Here are a few of the many methods in the `Calendar` class.

```
public static Calendar getInstance()
public Date getTime()
public int get(int field)
public void set(int field, int value)
public void set(int year, int month, int day)
```

Recall that using `static` as a modifier on a method such as `getInstance` means that the method is not invoked on a particular object. Instead a static method is invoked using the class name. In our case we would use `Calendar.getInstance()`. This is like using `Math.sqrt(2)` in the `Math` class. Therefore, to construct a `Calendar` object for the current locale and the current time we would use a statement such as

```
Calendar now = Calendar.getInstance();
```

[2]See the Java documentation for more examples.

The `getTime` method can be used to convert the current calendar to a `long` integer time.

Since there are so many instance data fields to "get" the designers opted for one "get" method and an integer value to indicate what field to get rather than a separate method for each field. For example, the field number for the year is the constant `Calendar.YEAR`. There are similar constants for the many other fields. Static constants in another class are always accessed using the syntax

ClassName`.`*constantName*

There are two forms of the `set` method. The first can set an individual field to a value. For example to set the day of the month field to 1 (first day) we could use

```
calendar.set(Calendar.DAY_OF_MONTH, 1);
```

The second form can be used to set the three date fields simultaneously. For example to get a calendar for December 25 in the current year use

```
Calendar christmas = Calendar.getInstance();
christmas.set(christmas.get(Calendar.YEAR), Calendar.DECEMBER, 25);
```

■ EXAMPLE 4.28 **(Using the `Calendar` class)** Try the following statements in BeanShell

```
bsh % import java.util.Date;
bsh % import java.util.Calendar;
bsh % Calendar now = Calendar.getInstance();
bsh % Date time = now.getTime();
bsh % print(time);
Sun Jun 15 10:04:12 EDT 2003
bsh % print(now.get(Calendar.YEAR));
2003
bsh % print(now.get(Calendar.MONTH)); // January is month 0
5
bsh % print(now.get(Calendar.DAY_OF_MONTH)); // June 15
15
bsh % print(now.get(Calendar.DAY_OF_WEEK));  // Sunday = 1
1
bsh % print(now.get(Calendar.DAY_OF_YEAR));
166
bsh % Calendar christmas = Calendar.getInstance();
bsh % int year = christmas.get(Calendar.YEAR);
bsh % christmas.set(year, Calendar.DECEMBER, 25);
bsh % print(christmas.getTime());
Thu Dec 25 11:13:01 EST 2003
```

This shows that the month numbers begin at 0 not 1 (June is month 5) and the days of the week begin at 1 (Sunday). ■

■ EXAMPLE 4.29 **(Leap years)** The following BeanShell statements

```
bsh % import java.util.Calendar;
bsh % Calendar feb2003 = Calendar.getInstance();
bsh % feb2003.set(2003, Calendar.FEBRUARY, 1);
bsh % Calendar feb2004 = Calendar.getInstance();
bsh % feb2004.set(2004, Calendar.FEBRUARY, 1);
bsh % print(feb2003.getActualMaximum(Calendar.DAY_OF_MONTH));
28
bsh % print(feb2004.getActualMaximum(Calendar.DAY_OF_MONTH));
29
```

show that the `getActualMaximum` method properly accounts for leap years. ■

`Person` class that uses `Calendar`

We can easily use the `Calendar` class to determine how old a person is this year. Your employer could use it to find out when you should retire. Given that `birthYear` is the year of birth and `age` is the age this year in years the following statements calculate the age:

```
Calendar now = Calendar.getInstance();
age = now.get(Calendar.YEAR) - birthYear;
```

Here is a simple class that uses these statements:

Class `Person`

book-projects/chapter4/calendar

```
package chapter4.calendar; // remove this line if you’re not using packages
import java.util.Calendar;

/**
 * A simple class that represents a person by a name and year of birth
 * There is also a method to determine how old the person is this year.
 */
public class Person
{
   private String name;
   private int birthYear;

   /**
    * Construct object given name and year of birth.
    * @param name the name of the person
    * @param birthYear the year of birth
    */
   public Person(String name, int birthYear)
   {
      this.name = name;
      this.birthYear = birthYear;
   }
   /**
    * Return the name.
    * @return the name
    */
   public String getName()
   {
      return name;
   }

   /**
    * Return the birth year
    * @return the birth year
    */
   public int getBirthYear()
   {
      return birthYear;
   }

   /**
    * Return the age this year.
    * @return the age this year
    */
   public int age()
   {
      Calendar now = Calendar.getInstance();
      return now.get(Calendar.YEAR) - birthYear;
   }
}
```

Here we have not included the age as a private data field. We could have done this and calculated the age in the constructor. However, if the age is calculated in the constructor it will never change subsequently and there is a remote possibility that an object of this class exists for a few years and the age will be incorrect.

Specialized `Calendar` class

Some classes like `Calendar` are very complex and for specialized applications it might be useful to develop a simpler version of the class. This is called adapting a class and the simpler version is called an **adapter class**.

If we want to write a program that displays the calendar for a given month then we do not need the full complexity of the `Calendar` class:

1. The only parts of a date we need are the year and month.
2. We need the day of the week for the first day of the month. For example, the first day of the month might be a Thursday so we need to skip Sunday to Wednesday and number days from 1 beginning on Thursday.
3. We need to know the number of days in the month, properly accounting for February in a leap year.

4. We need the names of the months if we want to print headings for each month.

The `Calendar` class can easily be adapted for this purpose. To do this we develop a class called `CalendarMonth` that has the following public interface.

```
public class CalendarMonth
{
   public CalendarMonth() {...}
   public CalendarMonth(int year, int month) {...}
   public int getYear() {...}
   public int getMonth() {...}
   public int dayOfWeek() {...}
   public int daysInMonth() {...}
   public String monthName() {...}
   public String toString() {...}
}
```

Here the first constructor uses the year and month for today and the second one uses given values. The `dayOfWeek` method returns the day of the week for the first day of the month. The return value is in the range 1 (Sunday) to 7 (Saturday). The `daysInMonth` method returns the number of days in the month, properly accounting for February in a leap year. The `monthName` method returns one of the strings `January` to `December` and the `toString` method returns a string such as `June 2003` which could be used to print headings.

To implement the class we need the private data field

```
private Calendar calendar;
```

Now the constructors can simply create a `Calendar` object for the first day of the month and the methods can use methods from the `Calendar` class. Adapter classes are usually quite simple since all the work is being done by the adapted class using aggregation. Here is the complete class followed by an explanation of the constructors and methods.

Class `CalendarMonth`

book-projects/chapter4/calendar

```
package chapter4.calendar; // remove this line if you’re not using packages
import java.util.Calendar;
import java.text.SimpleDateFormat;

/**
 * A class that represents a month of a given year in a form suitable
 * for printing calendars.
 */
public class CalendarMonth
{
   private Calendar calendar;

   public static final int JANUARY   = Calendar.JANUARY;
   public static final int FEBRUARY  = Calendar.FEBRUARY;
   public static final int MARCH     = Calendar.MARCH;
   public static final int APRIL     = Calendar.APRIL;
   public static final int MAY       = Calendar.MAY;
   public static final int JUNE      = Calendar.JUNE;
   public static final int JULY      = Calendar.JULY;
   public static final int AUGUST    = Calendar.AUGUST;
   public static final int SEPTEMBER = Calendar.SEPTEMBER;
   public static final int OCTOBER   = Calendar.OCTOBER;
   public static final int NOVEMBER  = Calendar.NOVEMBER;
   public static final int DECEMBER  = Calendar.DECEMBER;

   public static final int SUNDAY    = Calendar.SUNDAY;
   public static final int MONDAY    = Calendar.MONDAY;
   public static final int TUESDAY   = Calendar.TUESDAY;
   public static final int WEDNESDAY = Calendar.WEDNESDAY;
   public static final int THURSDAY  = Calendar.THURSDAY;
   public static final int FRIDAY    = Calendar.FRIDAY;
   public static final int SATURDAY  = Calendar.SATURDAY;

   /**
    * Construct a calendar for this month.
    */
   public CalendarMonth()
   {
      calendar = Calendar.getInstance();
      calendar.set(Calendar.DAY_OF_MONTH, 1);
   }

   /**
    * Construct a calendar for given year, month.
    * @param year the year
    * @param month the month in the range 0 to 11
    */
   public CalendarMonth(int year, int month)
   {
      calendar = Calendar.getInstance();
      calendar.set(year, month, 1); // months begin at 0
   }

   /**
    * Return the year for this calendar.
    * @return the year for this calendar.
    */
   public int getYear()
   {
      return calendar.get(Calendar.YEAR);
   }

   /**
    * Return the month for this calendar.
    * @return the month for this calendar.
```

```
     */
    public int getMonth()
    {
       return calendar.get(Calendar.MONTH);
    }


    /**
     * Return the day of week for the first day of this month.
     * @return the day of week for the first day of this month
     * in range 1 to 7 where 1 = Sunday, 7 = Saturday
     */
    public int dayOfWeek()
    {
       return calendar.get(Calendar.DAY_OF_WEEK);
    }

    /**
     * Return number of days in the month.
     * @return number of days in month.
     * Leap years are taken into account
     */
    public int daysInMonth()
    {
       return calendar.getActualMaximum(Calendar.DAY_OF_MONTH);
    }

    /**
     * Return the month name
     * @return the month name
     */
    public String monthName()
    {
       SimpleDateFormat f = new SimpleDateFormat("MMMM");
       return f.format(calendar.getTime());
    }

    /**
     * Return a string representation of a month
     * @return a string representation of a month
     */
    public String toString()
    {
       // example: June 2003
       SimpleDateFormat f = new SimpleDateFormat("MMMM yyyy");
       return f.format(calendar.getTime());
    }
}
```

Explanation of the class

We define twelve static public constants for the months in the same range as the `Calendar` class and similarly seven constants for the day names. Since they are static constants there is only one set for all objects and they are referred to using names such as `CalendarMonth.JUNE`. Since the constants here are public we should have included a javadoc comment for each one of them.

For the default constructor we start with the date for today and set the day field to 1 for the first day of the month. Similarly, the second constructor uses the three argument form of the set method.

Since we set the calendar object to the first day of the month we simply use the expression

```
calendar.get(Calendar.DAY_OF_WEEK)
```

in the `dayOfWeek` method to return a number in the range 1 to 7 for the day of week for the first day of the month.

For the last day of the month we use the `getActualMaximum` method from Example 4.29 which returns the maximum value of any field so we apply it to the `DAY_OF_MONTH` field. The returned value properly accounts for leap years.

The month name can be obtained using a special format of `SimpleDateFormat` so we use it in the `monthName` method. Similarly we use another date format in the `toString` method.

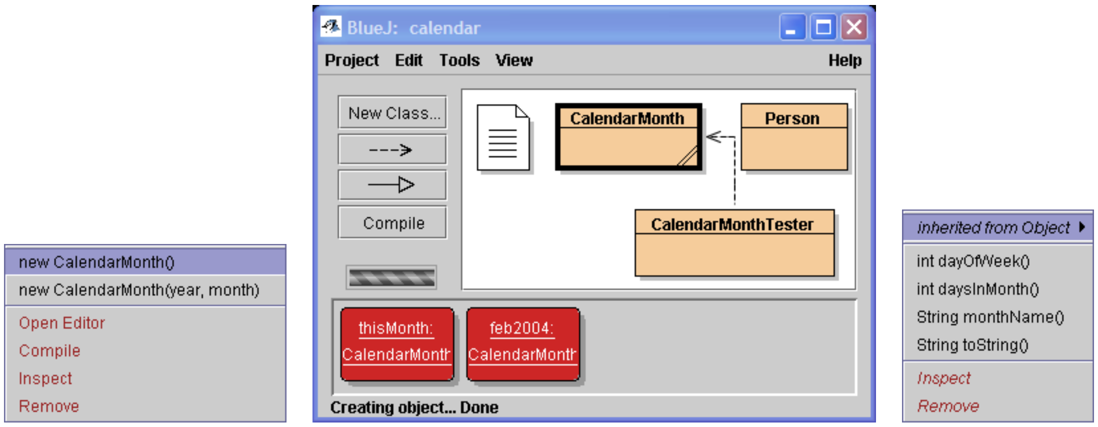
Testing the class

There is not much testing to do since our class is just a specialized case of the `Calendar` class.

The following example shows how to test the class in BeanShell.

■ EXAMPLE 4.30 **(Using `CalendarMonth` in BeanShell)**

```
bsh % addClassPath("c:/book-projects/chapter4/calendar");
bsh % CalendarMonth thisMonth = new CalendarMonth();
bsh % print(thisMonth.dayOfWeek());
1
bsh % print(thisMonth.daysInMonth());
30
bsh % print(thisMonth.monthName());
June
bsh % print(thisMonth);
June 2003
bsh % CalendarMonth feb2004 = new CalendarMonth(2004, CalendarMonth.FEBRUARY);
bsh % print(feb2004.dayOfWeek());
1
bsh % print(feb2004.daysInMonth());
29
bsh % print(feb2004.monthName());
February
bsh % print(feb2004);
February 2004
```



(a) (b) (c)

Figure 4.9: Testing `CalendarMonth` in `BlueJ`

In this test `thisMonth` refers to June, 2003. ■

To test the class using `BlueJ` follow the steps

1. Construct an object called `thisMonth` using the default constructor on the class menu (see Figure 4.9(a)).

2. Construct another object called `feb2004` using the other constructor on the class menu and enter 2004 for the year and `CalendarMonth.FEBRUARY` for the month. You now have two objects as shown in Figure 4.9(b).

3. Right click on the objects to get the object menu (see Figure 4.9(c)) and try the different methods.

4. If you use "inspect" on the object menu you will be able to see the static data fields by clicking on the "Show static fields" button in the Inspector window. These fields are also available by right clicking on the class rectangle.

Here is a short tester class that can also be used.

Class **`CalendarMonthTester`**

book-projects/chapter4/calendar

```
package chapter4.calendar; // remove this line if you're not using packages
/**
 * A simple test program for the CalendarMonth class
 */
public class CalendarMonthTester
{
   public CalendarMonthTester()
   {
   }

   public void doTest()
   {
      CalendarMonth thisMonth = new CalendarMonth();
      System.out.println("First day of month is " + thisMonth.dayOfWeek());
      System.out.println("Number of days in month is " + thisMonth.daysInMonth());
      System.out.println("Month name is " + thisMonth.monthName());
      System.out.println("Calendar name is " + thisMonth);

      System.out.println();
      CalendarMonth feb2004 = new CalendarMonth(2004, CalendarMonth.FEBRUARY);
      System.out.println("First day of month is " + feb2004.dayOfWeek());
      System.out.println("Number of days in month is " + feb2004.daysInMonth());
      System.out.println("Month name is " + feb2004.monthName());
      System.out.println("Calendar name is " + feb2004);
   }
}
```

4.5.2 Currency formatting

Another useful class in the `java.text` package is `NumberFormat` which can be used to format numbers as currency. The static method `getCurrencyInstance` constructs a currency format for your locale and returns a reference to it. Then you can use its `format` method to do the formatting. Therefore a currency formatter can be declared using

```
NumberFormat currency = NumberFormat.getCurrencyInstance();
```

This is the same idea as in the `Calendar` class. A static method is used instead of a constructor to create an object. The method has no arguments: everyone will get the appropriate formatter for their locale and programs will be locale independent.

For a North American locale this formatter rounds numbers to two decimal places, puts a dollar sign at the beginning of the number, and uses a comma as the thousands separator.

■ EXAMPLE 4.31 **(Formatting currency)** Try the BeanShell statements

```
bsh % import java.text.NumberFormat;
bsh % import java.util.Locale;
bsh % double salary = 100000.555;
bsh % NumberFormat currency = NumberFormat.getCurrencyInstance();
bsh % print(currency.format(salary));
$100,000.56
bsh % NumberFormat currencyCF =
NumberFormat.getCurrencyInstance(Locale.CANADA_FRENCH);
bsh % print(currencyCF.format(salary));
100 000,56 $
```

The default `currency` object for the author corresponds to `Locale.CANADA`. Other locales such as `Locale.CANADA_FRENCH` can be specified. In this case the dollar sign is at the end, the decimal point is replaced by a period and spaces are used instead of commas to separate thousands. ■

4.5.3 Formatting fixed and floating point numbers (Java 1.4)

The material in this section is not so important in Java 5 since the static `format` method in the `String` class and the `printf` method in `System.out` can more easily be used to format numbers (see Section 4.2.5).

Another useful class in the `java.text` package is `DecimalFormat` which can be used to format numbers in fixed or floating point (scientific) format. In scientific notation numbers are usually represented in the form $\pm m \times 10^e$ where $1 \leq m < 10$ is called the mantissa and e is called the exponent. Fixed format numbers are rounded to a fixed number of digits after the decimal point without an exponent and are suitable for numbers in a small range such as 1 to 10.

For example, to specify fixed notation with 1 or more digits to the left of the decimal point and five digits to the right you can use the format

```
DecimalFormat fix = new DecimalFormat(" 0.00000;-0.00000");
```

Here the format string comes in two parts. To the left of the semi-colon is the format for non-negative numbers and to the right is the format for negative numbers. Here we specify a leading space for non-negative numbers and a minus sign for negative numbers. This means that in a column numbers of mixed sign will line up.

The `format` method has the prototype

```
public String format(double d)
```

so `fix.format(d)` would be used to format a `double` number as a string using the specified format. Here is a simple example.

■ EXAMPLE 4.32 **(Fixed format)** Try the statements

```
bsh % import java.text.DecimalFormat;
bsh % DecimalFormat fix = new DecimalFormat(" 0.00000;-0.00000");
bsh % print(fix.format(Math.PI));
 3.14159
bsh % print(fix.format(-Math.PI));
-3.14159
```

The numbers are properly rounded. ■

Similarly, to specify a scientific format with one non-zero digit to the left of the decimal point, five digits after the decimal point and a three digit exponent we use the format

```
DecimalFormat sci = new DecimalFormat(" 0.00000E000;-0.00000E000");
```

Here is an example.

■ EXAMPLE 4.33 **(Scientific format)** Try the statements

```
bsh % DecimalFormat sci = new DecimalFormat(" 0.00000E000;-0.00000E000");
bsh % double d = 1.2345678E-23;
bsh % print(sci.format(d));
 1.23457E-023
bsh % print(sci.format(-d));
-1.23457E-023
```

The numbers are properly rounded. You can also use a lower case `e` for the exponent. ■

4.6 Review of OOP concepts

In this section we review and extend the basic OOP concepts of Chapter 3 and Chapter 4.

4.6.1 Constructing objects

In object-oriented programming the creation of objects is the most basic concept. Before an object can be used it must be constructed using a **constructor call expression** (Chapter 3, page 82) or by calling a static method in a class that returns an object of the class (factory method).

Using a constructor

Examples that use a constructor are

```
(1)   Circle c1 = new Circle(new Point(3,4), 5) ;
(2)   Point p = new Point(3,4) ;
(3)   Point q = new Point() ;
(4)   Circle c1 = new Circle(3, 4, 5) ;
(5)   BankAccount a = new BankAccount(123, "Reginald Hill", 4000) ;
(6)   SimpleDateFormat f = new SimpleDateFormat("MMMM yyyy") ;
```

In each case the constructor call expression is underlined. In the first example there are two constructor call expressions, one is used as an argument in the other.

Corresponding to each constructor call expression is a constructor prototype. For the above examples the prototypes are

```
public Circle(Point p, double radius)
public Point(double x, double y)
public Circle(double x, double y, double radius)
public BankAccount(int number, String name, double balance)
public SimpleDateFormat(String pattern)
```

A constructor prototype is the first line of the constructor declaration. It tells you how to write valid constructor call expressions. The general syntax for a constructor declaration was given in Chapter 3, Figure 3.28.

Using a static factory method

Examples of object construction that use a static method are

```
Calendar now = Calendar.getInstance() ;
NumberFormat currency = NumberFormat.getCurrencyInstance() ;
```

The **static method call expression** that constructs the object is underlined. Each such expression begins with the class name not an object name as in an instance method call expression (see Chapter 3, page 84). This particular kind of static method is often called a **factory method**, since its purpose is to manufacture a complex object such as a `Calendar` object.

Using `this` as a constructor call expression

We have seen two uses of `this`. It is used to access an instance data field. For example, in the `Point` class (page 119) we used `this.x` and `this.y` in the constructor to refer to the instance data fields because the constructor arguments had the same name.

Also, `this` can be used in a method call expression. For example, the `QuadraticRootFinder` class in Chapter 3, page 67 we used the method call expression `doCalculations()` in several places to aid in the calculations. We could have used `this.doCalculations()` to emphasize that the method is defined in "this" class although it was not necessary.

There is a third use of `this` which is quite useful in classes that contain several constructors. For example, in the `Circle` class we had three constructors

```
public Circle()
{
  center = new Point();
  radius = 1;
}

public Circle(double x, double y, double r)
{
   center = new Point(x, y);
   radius = r;
}

public Circle(Point p, double r)
{
   center = p;
   radius = r;
}
```

The first two constructors are really special cases of the third one so we could have written them as

```
public Circle()
{
   this(new Point(), 1);
}

public Circle(double x, double y, double r)
{
   this(new Point(x,y), r);
}
```

Here we are using `this` to call the third constructor. This does not seem like much of a simplification in this example, but it is in classes where constructor bodies contain lots of statements which would have to be duplicated.

Often there is a general constructor and all others are special cases. We can write the code once for the most general constructor and use `this` to refer to it in the other constructors.

Default constructor

In all the classes we have written we have included at least one constructor. It is common to see classes that have no constructors. For example in the `CircleTester` class (page 127) we included the no-arg constructor

```
public CircleTester()
{
}
```

which does nothing since there are no data fields in this class. The purpose of the class is simply to provide the `doTest` method that can be executed in BeanShell or BlueJ as a simple test of the `Circle` class. In fact this constructor could have been completely omitted from the class and when you right click on the `CircleTester` class rectangle you will still see the no-arg constructor on the menu.

The reason is that if no constructor declarations at all are present in a class the compiler will automatically provide a so-called **default no-arg constructor**. This is often humorously called the Miranda convention:

> *"You have the right to a constructor. If you do not have one, a default one will be provided for you by the compiler."*

The no-arg constructor, whether we provide it or let the compiler provide it, allocates memory for an object, like any constructor, but it also provides default initialization for any uninitialized data fields according to the following rules:

1. A value of 0 is assigned to all uninitialized numeric data fields.
2. A reference value of `null` is assigned to all uninitialized data fields of object type (see Section 4.6.2 below).

You can try it with BlueJ by writing the following three very simple classes in a project.

```
public class One
{
  // the simplest of all classes
}

public class Two
{
   private int k;
   private One one;
}

public class Three
{
   private int k;
   private One one;

   public Three()
   {
   }
}
```

Construct an object of each type and use “Inspect” on the object menu to see the data fields for objects of classes `Two` and `Three`. In either case you will see

```
private int k = 0;
private One one = null;
```

indicating that there is a default initialization.

Of course, it is best not to rely on default initialization and a better version of `Two` or `Three` would be

```
public class Four
{
   private int k;
   private One one;

   public Four()
   {
      k = 0;
      one = null;
   }
}
```

Here the initialization is explicit.

4.6.2 Object references

Constructing an object is a three step process for the Java run-time system.

1. Memory space is allocated for the object and its instance data fields.
2. A reference to the object is returned so that it can be located.
3. This reference is assigned as the value of an object reference variable.

Thus, an object is located using an **object reference**. Each object reference can be stored in a variable called an **object reference variable** whose value gives the location of the object. The name of this variable is called the object name.

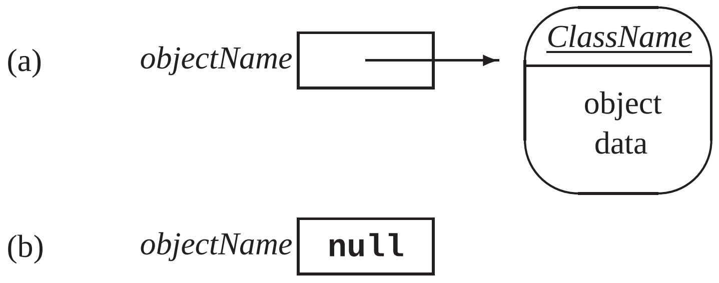


Figure 4.10: Box and arrow representation of an object and a reference to it.

A picture of an object and a reference to it is shown in Figure 4.10(a). This is often called the **box and arrow** notation. The object and its data fields are shown as a rounded box. The reference variable is shown as a square box with an arrow pointing to the object to symbolize the reference to the object, and the picture corresponds to the statement

ClassName objectName = `new` *ClassName* (*actualArguments*);

The right side is the constructor call expression. When it is executed the memory space for the object is obtained and a reference to the object is returned. This reference is then assigned as the value of the object reference variable *objectName*. For example,

```
Circle c = new Circle(3, 4, 5);
```

Object types are quite different than the primitive types such as `int` and `double`. This can be seen by comparing Figure 4.10 with Figure 2.1 and Figure 2.2. However the reference variable itself acts like a primitive type.

Null references

It is common to declare the name of an object without constructing the object right away. This situation is shown in Figure 4.10(b). Here we have no arrow. This corresponds to the declaration

ClassName objectName;

A reference that does not point to an object yet is called a **null reference** and is indicated by the Java keyword `null` so this declaration can be written as

ClassName objectName = `null`;

These two declarations are not quite the same. In the first case we have an uninitialized variable. In the second case the variable is initialized to the value `null`.

For example,

```
Circle c = null;
```

indicates that `c` is an object reference variable for and object of type `Circle` but it does not yet refer to any object. For such a variable it is a run-time error to invoke methods on a `null` reference. For example, the statements

```
Circle c = null;
double radius = c.getRadius();
```

would give an error called a `NullPointerException` (references are sometimes called pointers) because there is no object to invoke the `getRadius` method on. This is a common error that you must watch out for.

Comparison of primitive and reference types

We have encountered two types of variables in Java:

primitive types
numeric types such as `int`, `long`, `char`, `float`, and `double` introduced in Chapter 2 and the `boolean` type which will be introduced in a later Chapter.

reference types
object types such as `String`, `Circle`, and `BankAccount` introduced in Chapter 3 and Chapter 4.

These two types work in very different ways. For example, if `d` is a `double` variable then we cannot take its square root by doing something like `d.sqrt()`. Since `d` is not an object it has no methods. Instead we need to apply the static `Math.sqrt` method and use `Math.sqrt(d)`. But if `c` is a reference to a `Circle` object then we can use `c.getCenter()` to invoke the `getCenter` method on `c` and return the center of the circle. Thus, you cannot invoke methods on variables of primitive type.

The situation is shown in Figure 4.11. Here we see in part (a) that the value of the `double`

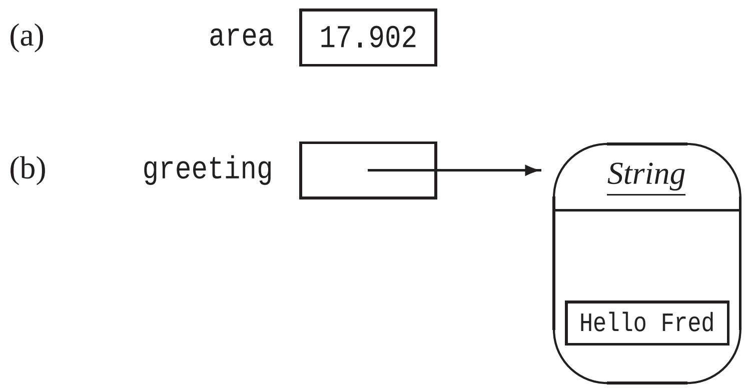


Figure 4.11: Comparison of primitive and reference types.

variable `area` is stored directly in the memory allocated for `area` but the value of `greeting` is a reference to the string object `"Hello Fred"` indicated by the arrow, and the reference is stored in the memory allocated for `greeting`. The `String` object, namely the string value `"Hello Fred"`, is stored elsewhere in memory reserved for the object. In fact as mentioned above we can think of the reference `greeting` as a primitive type – the memory address or location of the object.

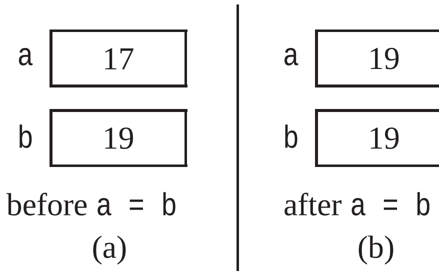


Figure 4.12: a = b for primitive types

Why do we need both primitive and reference types?

Why not use reference types for all variables? This could have been done in Java and is done in other OOP languages such as Smalltalk. However some types such as `int` and `double` were specified as primitive types to make their use more efficient as there is a certain amount of overhead in following a reference to find its object.

Assignment statements for reference types

Another major difference between primitive types and reference types occurs with the interpretation of assignment statements such as

```
a = b;
```

If `a` and `b` are variables of primitive types then we know this statement means to assign the value of `b` as the value of `a`. In other words `a` and `b` have the same values after the assignment. This is shown in Figure 4.12. In part (a), before the assignment `a = b`, variable `a` has the value `17`, and variable `b` has the value `19`. In part (b), after the assignment `a = b`, both variables have the same value `19`.

However, if `a` and `b` are references to objects then the situation is quite different, as shown in Figure 4.13. In part (a), before the assignment `a = b`, there are separate references for each of the objects *objectA* and *objectB* so `a` and `b` hold different values. In part (b), after the assignment `a = b`, `a` and `b` now hold the same values, namely a reference to *objectB*.

Unless there was some other reference to *objectA*, it has now become an orphan. This is indicated by the question mark beside *objectA*. The Java interpreter contains a garbage collector that will eventually delete it and reclaim the memory it uses.

■ EXAMPLE 4.34 **(`BankAccount` example of a = b)** For the `BankAccount` class the BeanShell statements

```
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount fred = new BankAccount(123, "Fred", 150.0);
bsh % BankAccount mary = new BankAccount(345, "Mary", 350.0);
bsh % mary = fred;
bsh % mary.withdraw(100.0);
```

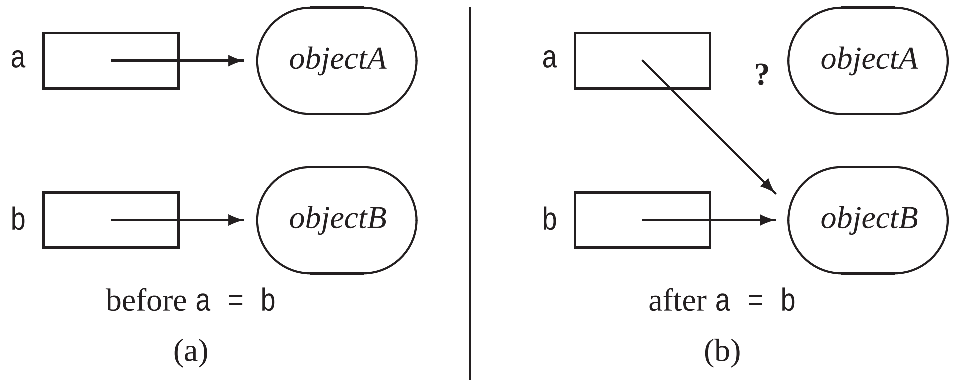


Figure 4.13: a = b for reference types

```
bsh % print(mary);
BankAccount[number=123, name=Fred, balance=50.0]
bsh % print(fred);
BankAccount[number=123, name=Fred, balance=50.0]
```

make both `mary` and `fred` refer to the same account, the one with account number `123` as shown by the `print` statements. “Mary” and “Fred” now have the same bank account, so `100` dollars is withdrawn from account number `123`. The other account, Mary’s original account with account number `345`, is an orphan and can no longer be referenced. ■

4.6.3 Using references as arguments and method return values

We have seen in Section 4.4.1 (page 117) on association that object references of one class can be used in another class in the following ways.

1. **local variables in the body of a constructor or method**
   For example, on page 118 we use

   ```
   TriangleCalculator tri = new TriangleCalculator(a,b,g);
   ```

   in the body of the `doTest` method in `TriangleCalculatorTester`.

2. **constructor or method arguments**
   For example, in `Circle` (page 122) we have the constructor prototype

   ```
   public Circle(Point p, double r)
   ```

   so the first argument is a reference to a `Point`.

3. **method return values**
   For example, in most of our classes we have used the `toString` method which returns a reference to a `String` object. As another example, the `getCenter` method in the `Circle` class contains the `return` statement

```
return center;
```

which returns a reference to the `Point` object for the center of the circle.

4. **instance data fields**
   This is aggregation and we have seen several examples. For example, `PasswordGenerator` (page 113) has the instance data field

   ```
   private Random random;
   ```

   and in `CalendarMonth` (page 133) we have the instance data field

   ```
   private Calendar calendar;
   ```

4.6.4 Data encapsulation and integrity

One of the benefits of object oriented programming is that data can be spread out over a number of objects. Each object is then responsible for its own data, independent of the data of any other object. This is called **data encapsulation**.

For example, in the `Point` class each object has its own copies of the data fields `x` and `y` representing the coordinates of the point. This particular class was designed to be immutable: it is not possible to change the coordinates of a `Point` object after it was constructed since there are no “set” methods and the data fields are private.

If possible it is a good idea to make your classes immutable. This is not possible in general. Many classes, such as `BankAccount` (page 106), must be mutable since the `withdraw` and `deposit` methods must change the account balance.

Having private data fields is a necessary condition for the immutability of a class. For example, in the `Point` class we could violate immutability in two ways:

- make the data fields `public` instead of `private`

  ```
  public double x;
  public double y;
  ```

- provide `setX` and `setY` methods:

  ```
  public void setX(double x)
  {
     this.x = x;
  }

  public void setY(double y)
  {
     this.y = y;
  }
  ```

With public data fields we can construct an object and then directly change its data fields as follows

```
Point = new Point(3,4);
p.x = 4; // 4 is new x coordinate
p.y = 5; // 5 is new y coordinate
```

If you try to do this with `private` data fields the compiler will give you an error message.

Similarly, if the "set" methods are included but the data fields are still `private` then the data fields can be changed indirectly as follows

```
Point = new Point(3,4);
p.setX(4); // 4 is new x coordinate
p.setY(5); // 5 is new y coordinate
```

Later we will see that if we need to make the class mutable the "set" methods are preferable to making the data fields public.

The same ideas apply to the `BankAccount` class. To maintain data integrity it would be necessary to ensure that an account cannot have a negative balance, and that the `withdraw` method cannot withdraw more money than is available. Our version of the class does not ensure these conditions but we will be able to modify the class when we learn about conditional statements.

Side-effects

When a class is associated with a mutable class it is possible to have undesirable side-effects that violate data encapsulation. To see this we will write a new mutable version of the `Point` class called `MPoint` that contains `setX` and `setY` methods[3]. The class is in a BlueJ project called `book-projects/chapter4/side_effects`.

Class `MPoint`

book-projects/chapter4/side_effects

```
package chapter4.side_effects; // remove this line if you're not using packages

/**
 * This class is like Point except it is a mutable version
 * with setX and setY methods. We have also added a copy constructor.
 */
public class MPoint
{
   private double x;
   private double y;

   /**
    * Construct a point from its coordinates.
    * @param x the x coordinate of the point
    * @param y the y coordinate of the point
```

[3]We have also added a copy constructor to be discussed below

```
 */
public MPoint(double x, double y)
{
  this.x = x;
  this.y = y;
}

/**
 * Construct the default point (0,0).
 */
public MPoint()
{
   x = 0.0;
   y = 0.0;
}

/**
 * Copy constructor
 * @param p point to copy
 */
public MPoint(MPoint p)
{
   x = p.x;
   y = p.y;
}

/**
 * Return the x coordinate of this point.
 * @return the x coordinate of this point
 */
public double getX()
{
   return x;
}

/**
 * Return the y coordinate of this point.
 * @return the y coordinate of this point
 */
public double getY()
{
   return y;
}

/**
 * Set a new value for the x coordinate
 * @param x the new x coordinate
 */
public void setX(double x)
{
   this.x = x;
}
```

```
   /**
    * Set a new value for the y coordinate
    * @param y the new y coordinate
    */
   public void setY(double y)
   {
      this.y = y;
   }

   /**
    * Return a string representation of an MPoint object.
    * @return a string representation of an MPoint object
    */
   public String toString()
   {
      return "MPoint[" + x + ", " + y + "]";
   }
}
```

In the same project we also write the following version of the `Circle` class called `MCircle` that uses `MPoint` instead of `Point` and violates data encapsulation.

Class `MCircle`

book-projects/chapter4/side_effects

```
package chapter4.side_effects; // remove this line if you're not using packages

/**
 * This class is identical to Circle except it used MPoint, the mutable point
 * class, instead of Circle, which uses the immutable Point class.
 */
public class MCircle
{
   private MPoint center;
   private double radius;

   /**
    * Construct circle with given center point and radius.
    * @param p the center of the circle
    * @param r the radius of the circle
    */
   public MCircle(MPoint p, double r)
   {
      center = p;
      radius = r;
   }

   /**
    * Construct circle with given center coordinates and radius.
    * @param x the x coordinate of the circle center
    * @param y the y coordinate of the circle center
    * @param r the radius of the circle
    */
   public MCircle(double x, double y, double r)
   {
      center = new MPoint(x,y);
      radius = r;
   }

   /**
    * Construct a default circle: a unit circle with center (0,0)
    * and radius 1.
    */
   public MCircle()
   {
      center = new MPoint();
      radius = 1;
   }

   /**
    * Return radius of circle.
    * @return radius of circle
    */
   public double getRadius()
   {
      return radius;
   }

   /**
    * Return center of circle.
    * @return center of circle
    */
   public MPoint getCenter()
   {
      return center;
   }

   /**
    * Return a string representation of an MCircle object.
    * @return a string representation of an MCircle object
    */
   public String toString()
   {
      return "MCircle[" + center + ", " + radius + "]";
   }
}
```

The following example illustrates the side-effects.

■ EXAMPLE 4.35 **(An undesirable side-effect)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter4/side_effects");
bsh % MPoint p = new MPoint(3,4);
bsh % MCircle c = new MCircle(p, 5);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
bsh % p.setX(999);
bsh % print(c);
MCircle[MPoint[999.0, 4.0], 5.0]
```

and notice that the change of the `x` coordinate of `p` from `3` to `999` has miraculously changed the `x` coordinate of the center of the circle to `999` too.

Why should a change in a point that should have nothing to do with the circle, change the circle object? The reason is that `p` and the instance data field `center` both reference the same object since the constructor contains the statement

```
center = p;
```

so any change in this object through `p` is also a change in the center of the circle. ■

You can also confirm these results using `BlueJ` as follows

1. Construct an `MPoint` object called `p` with coordinates `3` and `4`.
2. Construct an `MCircle` object called `c` using the constructor that has an `MPoint` object and choose `p` as the `MPoint` object and a radius of 5.
3. Now use the `setX` method on the object menu of `p` and change the `x` coordinate to `999`.
4. Finally, use the inspector choice on object menu for `c` and examine the object reference `center` to see that the `x` coordinate of the center is now `999`

The following example shows another way to violate data encapsulation with `MCircle`.

■ EXAMPLE 4.36 **(Another side-effect)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter4/side_effects");
bsh % MCircle c = new MCircle(3,4,5);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
bsh % MPoint p = c.getCenter();
bsh % p.setX(999);
bsh % print(c);
MCircle[MPoint[999.0, 4.0], 5.0]
bsh %
```

and notice that we can use `getCenter` to obtain a reference to the center point. Then we can use this reference to change the `x` coordinate from `3` to `999`. The reason here is the statement

```
return center;
```

in the `getCenter` method, which also returns a reference to `p`. ■

These situations involving undesirable side-effects are shown in Figure 4.14. Here it is clear that there is only one `MPoint` object and it can be modified either through reference `p` or reference `center`.

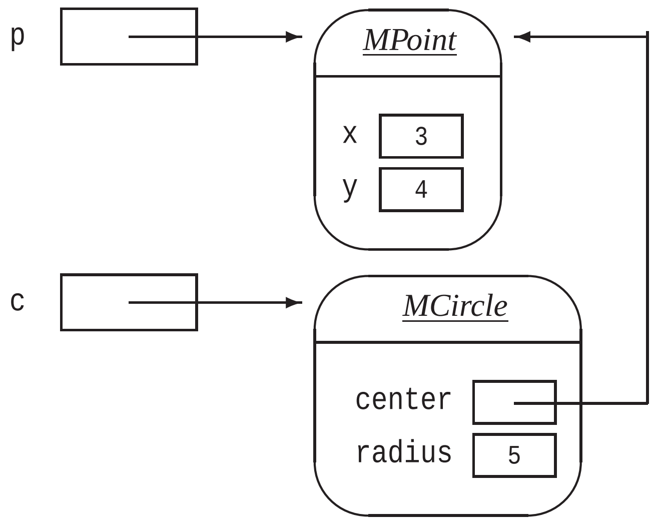


Figure 4.14: A side-effect after `p` changes the `x` from `3` to `999`
.

To see that this side-effect does not occur in the original versions of these classes try the following example in BeanShell.

■ EXAMPLE 4.37 **(No side-effect in original classes)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter4/geometry");
bsh % import Point; // necessary or we get java.awt.Point
bsh % Point p = new Point(3,4);
bsh % Circle c = new Circle(p, 5);
bsh % print(c);
Circle[Point[3.0, 4.0], 5.0]
bsh % p = new Point(999,4);
bsh % print(c);
Circle[Point[3.0, 4.0], 5.0]
```

and notice now that there is no change to the center coordinates of the circle. This occurs because `Point` is an immutable class so there is no way to change the `Point` object that was created. All we can do is create a new object for `p` to reference in the statement

```
p = new Point(999,4);
```

So now instead of having two references `p` and `center` to the same object we have references to different objects. Now `center` is a reference to $(3,4)$ and the `p` is a reference to $(999,4)$. ■

The same analysis applies to the `String` class since it is also immutable. As a general rule it is best to make classes immutable if possible.

Before fixing `MCircle` to remove side effects we discuss the concept of a **copy constructor**.

Copy constructor

A copy constructor makes a copy of the object referenced by its argument. For a class called *ClassName* the copy constructor has the prototype

> `public` *ClassName* `(`*ClassName objectName*`)`

We have included the copy constructor

```
public MPoint(MPoint p)
{
   x = p.x
   y = p.y
}
```

in the `MPoint` class. It is legal to use `p.x` and `p.y` here even though the data fields are private because we are inside the `MPoint` class itself and instance data fields can be accessed directly in this case.

Using the copy constructor we can modify the `MCircle` class so that side-effects do not occur. The first modification is to change the constructor that accepts an `MPoint` reference as an argument to

```
public MCircle(MPoint p, double r)
{
   center = new MPoint(p);
   radius = r;
}
```

This makes a copy of the `MPoint` object referenced by `p` and assigns its reference to `center`. Now the `MCircle` object has its own copy of this object which is not affected by any changes to the callers object referenced by `p`.

The second modification is to change the `getCenter` method to

```
public MPoint getCenter()
{
   return new MPoint(center);
}
```

Again, this returns a reference to a copy of the center object not a reference to the center object itself. In other words, by making copies we do not give away to the caller a reference to our private data field of type `MPoint`.

These changes are tested in a project called `book-projects/chapter4/no_side_effects` that contains our `MPoint` class and the following version of `MCircle`.

Class `MCircle`

book-projects/chapter4/no_side_effects

```
package chapter4.no_side_effects; // remove this line if you’re not using packages

/**
 * This class is like the original MCircle class
 * but we have made it immutable so there are no side-effects.
 */
public class MCircle
{
   private MPoint center;
   private double radius;

   /**
    * Construct circle with given center point and radius.
    * @param p the center of the circle
    * @param r the radius of the circle
    */
   public MCircle(MPoint p, double r)
   {
      center = new MPoint(p);
      radius = r;
   }

   /**
    * Construct circle with given center coordinates and radius.
    * @param x the x coordinate of the circle center
    * @param y the y coordinate of the circle center
    * @param r the radius of the circle
    */
   public MCircle(double x, double y, double r)
   {
      center = new MPoint(x,y);
      radius = r;
   }

   /**
    * Construct a default circle: a unit circle with center (0,0)
    * and radius 1.
    */
   public MCircle()
   {
      center = new MPoint();
      radius = 1;
   }

   /**
    * Return radius of circle.
    * @return radius of circle
    */
   public double getRadius()
   {
      return radius;
   }

   /**
    * Return a copy of the center of circle.
    * @return a copy of the center of circle
    */
   public MPoint getCenter()
   {
      return new MPoint(center);
   }

   /**
    * Return a string representatio of an MCircle object.
    * @return a string representation of an MCircle object
    */
   public String toString()
   {
      return "MCircle[" + center + ", " + radius + "]";
   }
}
```

Now run the BeanShell test in Example 4.35 using this version of `MCircle`:

■ EXAMPLE 4.38 **(The side-effects are gone)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter4/no_side_effects");
bsh % MPoint p = new MPoint(3,4);
bsh % MCircle c = new MCircle(p, 5);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
bsh % p.setX(999);
bsh % print(c);
MCircle[MPoint[3.0, 4.0], 5.0]
```

and the side-effects are gone: changing the `x` coordinate does not change the center of the circle. ■

With the immutable version of `MCircle` we have the situation shown in Figure 4.15. Now there are two `MPoint` objects instead of one. Side-effects like this always arise from mutable classes and references and cannot occur for primitive types such as `int` or `double`.

`BankAccount` example

As a simple example of association and aggregation that has desirable side-effects consider a class called `TransferAgent` whose objects can take two `BankAccount` and transfer a given amount of money from one account to the other. The public class interface is given by

```
public class TransferAgent
{
```

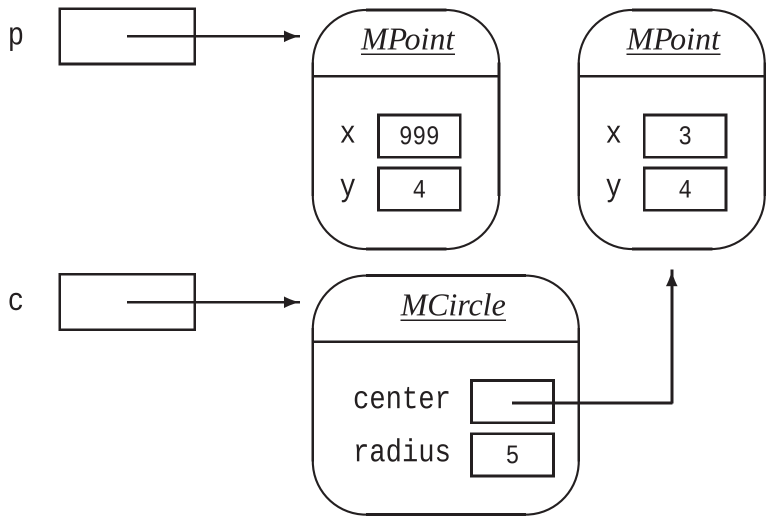


Figure 4.15: The side-effect disappears after `p` changes `x` from `3` to `999`.

```
      public TransferAgent(BankAccount from, BankAccount to) {...}
      public void transfer(double amount) {...}
   }
```

Here the constructor arguments specify references to the two accounts and the `transfer` method does the transfer of a specified amount from the `from` account to the `to` account. Therefore the `TransferAgent` is associated with (or uses) the `BankAccount` class.

The class implementation needs two private data fields of type `BankAccount` to hold references to the two accounts that are involved in the transfer. Therefore the class also illustrates aggregation. The constructor will initialize these fields. The `transfer` method just needs to withdraw the given amount from one account and deposit it in the other account using the `withdraw` and `deposit` methods. Here is the complete class.

Class `TransferAgent`

book-projects/chapter4/bank_account

```
package chapter4.bank_account; // remove this line if you're not using packages
/**
 * A TransferAgent object can be used to transfer a given amount of money
 * from one account to another.
 */
public class TransferAgent
{
   private BankAccount from;
   private BankAccount to;

   /**
    * Construct an object for two accounts
    * @param from the "from" account
    * @param to the "to" account
    */
   public TransferAgent(BankAccount from, BankAccount to)
   {
      this.from = from;
      this.to = to;
   }

   /**
    * Transfer the given amount of money from one account to the other.
    * @param amount amount to transfer
    */
   public void transfer(double amount)
   {
      from.withdraw(amount);
      to.deposit(amount);
   }
}
```

There are side-effects since the `BankAccount` class is mutable and the `transfer` method will change the balance data field of the two `BankAccount` objects given as arguments in the constructor. However, here the account objects are changed from inside the class, by the `transfer` method, not from outside it as in the `MCircle` Example 4.35, where the center of the circle was changed from outside the class. In fact the purpose of the `transfer` method is to produce these side-effects. You can try this class in BeanShell or BlueJ.

■ EXAMPLE 4.39 **(Desirable side-effects)** The statements

```
bsh % addClassPath("c:/book-projects/chapter4/bank_account");
bsh % BankAccount fred = new BankAccount(123, "Fred", 1000);
bsh % BankAccount mary = new BankAccount(345, "Mary", 1000);
bsh % TransferAgent agent = new TransferAgent(fred, mary);
bsh % agent.transfer(500);
bsh % print(fred);
BankAccount[number=123, name=Fred, balance=500.0]
bsh % print(mary);
BankAccount[number=345, name=Mary, balance=1500.0]
```

transfer $500 from the `fred` account to the `mary` account. ■

In BlueJ try the following steps.

1. Create two `BankAccount` objects with the account numbers, names, and initial balances given above in the BeanShell example.
2. Create a `TransferAgent` object called `agent` for the two accounts as given in the above BeanShell example. The result is shown in Figure 4.16.
3. From the `agent` object menu choose transfer and enter 500 as the amount.
4. Inspect the two account objects or use their `getBalance()` methods to verify that the transfer was made.

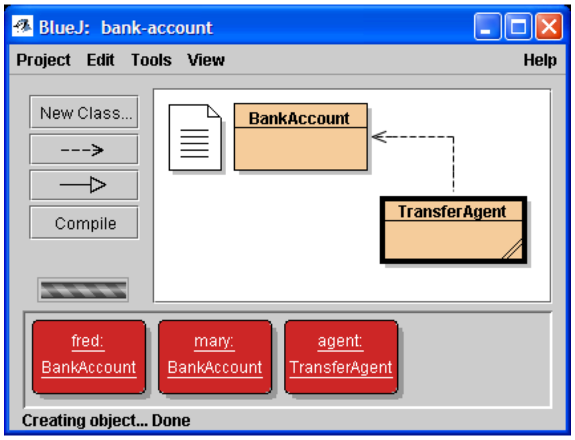


Figure 4.16: The `TransferAgent` example in BlueJ

4.6.5 Instance variables and methods

We now summarize the concepts related to instance variables and methods.

Instance variables

The instance variables (instance data fields) of a class, if any, are declared in the class but outside any method or constructor. Each object from the class has its own set of these instance variables. Each instance method has access to these variables to examine or change the state of the object.

Instance methods

The methods we have written so far have been methods that we can invoke on an object. We have called these methods **instance methods** or **object methods**. In BlueJ they are accessed by right clicking on an object and choosing a method from the object menu.

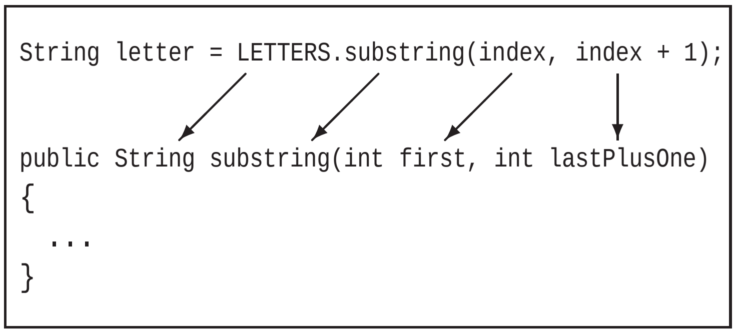
In Java every method belongs to some class and when we construct an object from a class we often say that the object is an instance of the class. Each class we use or write usually has one or more instance methods. These methods define the behavior of objects from the class. A template for a method declaration was given in Chapter 3, Figure 3.29.

Using an instance method

To use a method means to invoke it on an object. For example, the `getArea` method in the `CircleCalculator` class in Chapter 3 (page 106), returns the area of the circle so we say that a `CircleCalculator` object knows how to compute the area of a circle – this is part of its behavior.

When an instance method is executed it is associated with some object. For example, if `circle` is a `CircleCalculator` object then the statement

```
double area = circle.getArea();
```



```
String letter = LETTERS.substring(index, index + 1);

public String substring(int first, int lastPlusOne)
{
  ...
}
```

Figure 4.17: Correspondence between call expressions and prototypes

calculates the area and assigns it to variable `area`. The expression `circle.getArea()` is called an **instance method call expression**, and we say that we are invoking the `getArea` method on the `circle` object. We can also say that we are **sending a message** to an object.

Here are some examples we have used in Chapter 4 with each instance method call expression underlined.

```
(1)  int length = name.length() ;
(2)  char first = test.toUpperCase().charAt(0) ;
(3)  password = password + LETTERS.substring(index, index + 1) ;
(4)  String upper = test.toUpperCase() ;
(5)  account.deposit(100)) ;
(6)  String firstInitial = firstName.substring(0,1).toUpperCase() ;
(7)  double x = c2.getCenter().getX() ;
(8)  doCalculations() ; // this is the implied object
(9)  long t = now.getTime() ;
(10) String n3 = f3.format(now) ;
```

There is a direct correspondence between a method call expression and the method prototype as shown by the example in Figure 4.17.

In the simplest case an instance method call expression has one of the forms (see Chapter 3, page 84)

objectName`.`*methodName*`(`*actualArguments*`)`
`this.`*methodName*`(`*actualArguments*`)`

The second form refers to a method in its own class and the “`this.`” part can be omitted.

Examples (2), (6), and (7) illustrate an important idea called **message composition**. For example, in (2) we first send the `toUpperCase` message to the string `test`. This returns another string and we send the `charAt(0)` message to it to finally obtain an upper case version of the first character of the string `test`.

Similarly, in (7) we send the `getCenter()` message to the point `c2`. This returns a point and we send the `getX()` method to it to finally obtain the `x` coordinate of the center of the circle.

Method composition

Instance message composition can be extended to several steps so the general form is

> *objectName.methodName(args1)* `.` *methodName(args2) ... methodNameN(argsN)*
> `this.`*methodName(args1)* `.` *methodName(args2) ... methodName(argsN)*

As mentioned above "`this.`" can be omitted. Read these expression from left to right and as long as each expression returns the right kind of object the entire expression will make sense.

4.6.6 Static variables, constants, and methods

Static variables

Variables can be declared static using the `static` modifier on the variable declaration. We have not used static variables yet but an example we will use below is

```
private static int count = 0;
```

which declares `count` to be a static variable whose initial value is 0.

A **static variable** is quite different from an instance variable where each object has its own copy of the variable. This is shown in Figure 4.15 where the two `MPoint` objects clearly have their own copies of the `x` and `y` data fields. For a static variable there is only one copy and all objects of the class share this variable – any constructor or method in the class can inspect or modify it. In the above example any change in the value of `count` made by some object will also be seen by all other objects from the class. We say that `count` is a **shared variable**.

Static constants

A static constant in a class is declared in the same way as a static variable but the `final` keyword is included. Constants in a class are normally declared to be `static` since there is no need for each object to have its own copy of a constant. The constant `Math.PI` is an example from the `Math` class and we have also introduced some examples of our own. In the `PasswordGenerator` class (page 113) we defined a string of lower case letters as a static constant:

```
public static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
```

and in the `CalendarMonth` class we used static integer constants such as

```
public static final int JANUARY = Calendar.JANUARY;
```

Static methods

The `static` modifier is used on a method prototype and declaration to distinguish a static method from an instance method. We have not written a static method yet but we have used some. A static method in a class is not associated with any object of the class. Thus, the statements in the body of a static method cannot refer to any of the instance data fields.

Using static methods

The syntax for using a static method is similar to that of an instance method except the class name is used in place of the object name. For example, all the methods in the `Math` class are static methods (see Chapter 2) so to calculate $\sqrt{2}$ and assign it to a variable we would use

```
double root2 = Math.sqrt(2.0);
```

The expression `Math.sqrt(2.0)` is called a **static method call expression**. The class name `Math` must be present so the compiler will know which class contains the `sqrt` method. The `Math` class is an example of a class that contains only static methods.

Here are some examples we have used in Chapter 3 and Chapter 4 with each static method call expression underlined.

(1) `beta = Math.toDegrees(beta);`
(2) `System.out.println("Area:  " + area);`
(3) `Calendar now = Calendar.getInstance();`
(4) `NumberFormat currency = NumberFormat.getCurrencyInstance();`

Examples like (2) are interesting. Here `System` is the name of a class so `System.out` is a static variable in this class. The type of this variable is `PrintWriter` which has several `print` and `println` instance methods.

The methods in the static method call expressions in (3) and (4) are called **factory methods** because they are static methods in a class whose job it is to construct an object from some class and return a reference to it.

Thus, the general syntax for invoking a static method is

ClassName`.`*methodName*`(`*actualArguments*`)`
methodName`(`*actualArguments*`)`

The second form would correspond to a static method in its own class.

Counting the number of objects created from a class

For the `Point` class (page 119) suppose that we want to count how many objects from the class have been instantiated (created). We can do this with a static variable and a static method using the project called `book-projects/chapter4/static_variable`. that contains a modified version of the `Point` class from project `book-projects/chapter4/geometry`. The new `Point` class is obtained by adding the following `static` variable to the data field section (outside any constructor or method).

```
private static int count = 0;
```

Now add the following statement to the body of each constructor;

```
count++;
```

Thus, each time an object is created `count` will be incremented.

Finally, add the following static method to the class

```
public static int getCount()
{
   return count;
}
```

Testing the class with BeanShell Now you can test the class in BeanShell as follows.

■ EXAMPLE 4.40 **(Counting the number of objects)** The statements

```
bsh % addClassPath("c:/book-projects/chapter4/static_variable");
bsh % import Point; // necessary of we get java.awt.Point
bsh % print(Point.getCount());
0
bsh % Point p1 = new Point(1,2);
bsh % Point p2 = new Point(3,4);
bsh % Point p3 = new Point(4,5);
bsh % Point p4 = p1;
bsh % int objectCount = Point.getCount();
bsh % print(objectCount);
3
```

show how to use the static method to count the number of `Point` objects that have been created. If you were expecting 4 as an answer instead of 3 recall from Figure 4.13, page 147, that `p1` and `p4` are references to the same object, the `Point` object with coordinates $(1,2)$. ■

Testing the class with BlueJ You can also use BlueJ to do the same test:

1. Create a few `Point` objects.
2. Now you will not find the `getCount` method on the object menu since it is not an instance method. Instead right click on the `Point` class box and it will be there.
3. Call the `getCount` method to display the number of objects created and displayed so far.

A class can contain a mixture of instance methods and static methods[4]. or it can contain only methods of one kind. This is often a design decision. One possible advantage of a static method is that it is not necessary to first construct an object before using the method. Only the class name is needed. Of course, to execute an instance method it is always necessary to construct an object first.

The `Math` class is an example of a class containing only static methods. If the designer had used instance methods then it would always be necessary to construct some "math" object before using one of the math functions and this does not make sense.

Some classes have both kinds of methods. For example, the `NumberFormat` class (see Section 4.5.2) has the static method `getCurrencyInstance` but it also has the instance method `format`.

[4]The `String` class is an example

The classes we have written so far have been classes containing only instance methods. If an instance method does not access any instance data fields then we could choose to make it an instance method or a static method. For example, the `doTest` method in the simple classes `TriangleCalculatorTester`, page 118, `CircleTester`, page 127, and `CalendarMonthTester`, page 137, do not access any data fields. In fact, these classes have no data fields. Therefore these methods could have been made static. The only difference in `BlueJ` is that we would now access the `doTest()` method from the class menu instead of the object menu.

4.6.7 Kinds of variables and arguments

So far we have seen four kinds of variables in Java.

- **Instance data fields:** They are instance variables that are declared outside any constructor or method in a class and can be accessed directly by any instance method in the class.
- **Static data fields:** They are static variables that are declared outside any constructor or method in a class and can be accessed by any instance or static method in the class.
- **Local variables:** They are variables declared in a method or constructor body and are only accessible within this body.
- **formal arguments:** They are declared in the constructor or method prototype and become local variables inside the constructor or method body.

4.6.8 Call by value argument passing mechanism

When a method or constructor call expression is executed, it is important to understand how the actual arguments in the expression are assigned to the formal arguments specified in the method or constructor declaration.

For example, in Figure 4.3 (page 105) there are three formal arguments that need to be matched with the three actual arguments in the constructor call expression. When the constructor is called the **values** of the actual arguments are assigned as the values of local variables with the names `accountNumber`, `ownerName` and `initialBalance`.

Thus, in the body of the constructor, `accountNumber` will have the value `123`, `ownerName` will have the value of a reference to the `String` object `"Peter Pascoe"`, and `initialBalance` will have the value `125.50`.

This is the **call by value** argument passing mechanism. The rules are simple:

- If an actual argument is a variable then a copy of its value is supplied as the value of the formal argument.
- If an actual argument is a literal (literal string or number for example) then this literal value is supplied as the value of the formal argument.
- If an actual argument is an expression then the expression is evaluated and its value is supplied as the value of the formal argument.

Call by value for an argument of primitive type

A consequence of call by value is that if a constructor or method argument is a primitive type and a variable is used as an actual argument, then the value of that variable can not be changed by the method since only a copy of the variable's value, not its location, is passed to the method. For example consider the following method, which could be a static or an instance method since it doesn't refer to the instance data fields of any class.

```
void addOne(int k)
{
   k = k + 1;
}
```

If the intent of this method is to add one to variable `k`, and somehow return it to the caller of the method, it does not work since `k` is a local variable which receives only a copy of the caller's value, not its location.

An interesting feature of BeanShell is that we can define this method without the keywords `static` or `public` and BeanShell will treat it as static method without having to specify to which class it belongs. Here is an example.

■ EXAMPLE 4.41 **(Arguments of primitive type)** The BeanShell statements

```
bsh % void addOne(int k) { k = k + 1; }
bsh % int m = 5;
bsh % addOne(m);
bsh % print(m);
5
bsh % print(k);
// Error: Undefined variable or class name, parameter: arg to method:
 print : at Line: 6 : in file: <unknown file> : print ( k )
```

do not change the value of `m` since the value of `m`, not its location, is assigned to `k` in the method body and 1 is added to the value of this local variable not to the callers variable `m`. When the method finishes executing the local variable `k` disappears and is no longer available. This feature of BeanShell is very convenient for testing static methods. ■

In BlueJ: we can write the small tester class

Class `ArgumentTester1`

book-projects/chapter4/arguments

```
package chapter4.arguments; // remove this line if you're not using packages

/**
 * Illustrating call by value for primitive types
 */
public class ArgumentTester1
{
   public void doTest()
   {
      int m = 5;
      addOne(m);
      System.out.println(m);
   }

   private static void addOne(int k)
   {
      k = k + 1;
   }
}
```

Now we can construct a `ArgumentTester1` object and invoke its `doTest()` method, which will call the static `addOne` method and display the resulting value of `m` which is still 5.

Call by value for an argument of reference type (object type)

For reference types the situation is quite different. We still use "call by value" but now the value passed is a copy of a reference to the caller's object so there will now be two references to the caller's object. Consider the following method.

```
void addOneDollar(BankAccount b)
{
   b.deposit(1.0);
}
```

Here `b` is a reference to a `BankAccount` object and this class is mutable, so the `deposit` statement will change the balance of the `BankAccount` object referenced by `b`. The following BeanShell example shows this. We have included a copy of `BankAccount` in the `arguments` project.

■ EXAMPLE 4.42 **(Arguments of reference type)** The BeanShell statements

```
bsh % addClassPath("c:/book-projects/chapter4/arguments");
bsh % void addOneDollar(BankAccount b) { b.deposit(1); }
bsh % BankAccount a = new BankAccount(123, "Fred", 1000);
bsh % addOneDollar(a);
bsh % print(a);
BankAccount[number=123, name=Fred, balance=1001.0]
```

show that the balance of the object referenced by `a` has $1 added to it because inside the method `b` is also a reference to this object. ■

This situation is illustrated in Figure 4.18. Before the `addOneDollar` method call, part (a), `a` is the only reference to the object. During the method call, part (b), `b` is a local variable that is a copy of `a` so it refers to the same object. After the method call `b` disappears and we have the original situation but with the balance increased by one dollar.

In BlueJ: we can write the small tester class

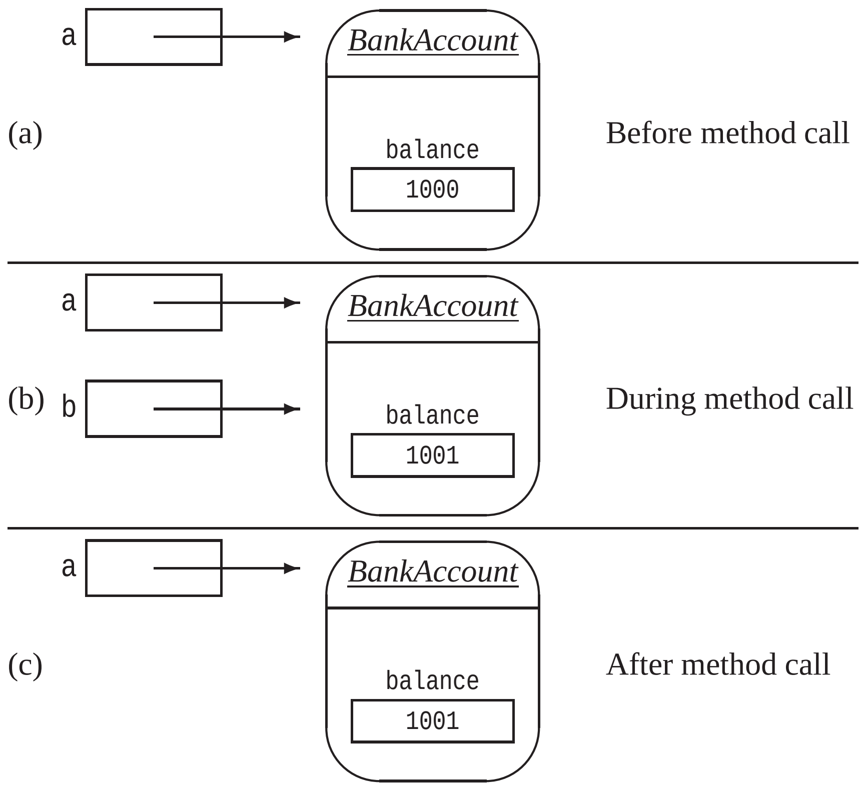


Figure 4.18: Call by value using references.

Class `ArgumentTester2`

book-projects/chapter4/arguments

```
package chapter4.arguments; // remove this line if you’re not using packages
import chapter4.bank_account.BankAccount; // remove this line if you’re not using packages

/**
 * Illustrating call by value for reference types
 */
public class ArgumentTester2
{
   public void doTest()
   {
      BankAccount a = new BankAccount(123, "Fred", 1000);
      addOneDollar(a);
      System.out.println(a);
   }

   private static void addOneDollar(BankAccount b)
   {
      b.deposit(1);
   }
}
```

We can construct a `Tester` object and invoke its `doTest()` method. This will call the static `addOneDollar` method and display the resulting value of `a` which will show the increase in balance by one dollar.

4.6.9 `main` method

So far our classes have been designed for execution inside the `BlueJ` environment but if you try to run one of them outside the `BlueJ` environment using the command line java interpreter you will get an error message indicating that the interpreter cannot find the infamous `main` method.

When the interpreter tries to run a class it looks for a special static method, called the `main`, method, having the exact form

```
public static void main(String[] args)
{
   ....
}
```

If found the interpreter will begin executing the statements in the body of this method. There are two ways to introduce the main method:

- Put a `main` method in the class you want to run. If your program consists of more than class one of them should have a `main` method and it will be called the main class.
- Write a special runner class that has a main method and run it with the interpreter. This has the advantage that the class you want to run doesn't need to be modified.

We illustrate both approaches and then show how to run classes containing a main method using a command prompt (Windows), or terminal window (Unix) and the java compiler (`javac`) and interpreter (`java`).

Adding a `main` method to a class

As an example, we make a new project called `main-method` containing `Point`, `Circle`, and `CircleTester` from project `geometry` using "Add class from file". The project is shown in Figure 4.19.

Now modify this version of `CircleTester` (page 127) by adding the main method

```
public static void main(String[] args)
{
   CircleTester tester = new CircleTester();
   tester.doTest();
}
```

to obtain the class

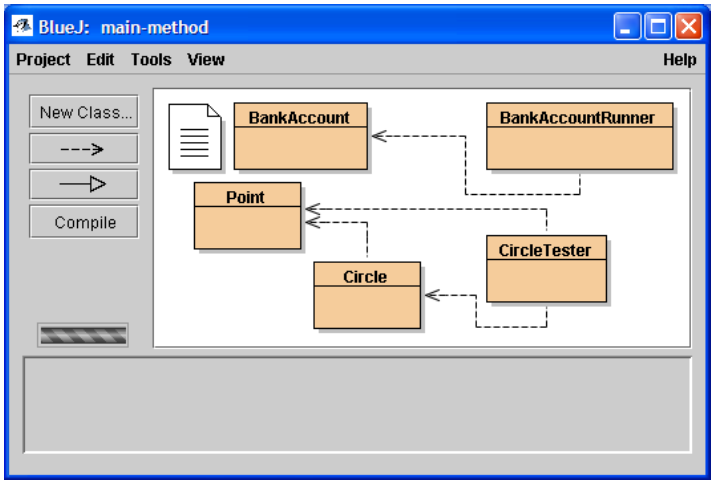


Figure 4.19: The `main-method` project.

Class `CircleTester`

book-projects/chapter4/main_method

```
package chapter4.main_method; // remove this line if you're not using packages
import chapter4.geometry.Point; // remove this line if you're not using packages
import chapter4.geometry.Circle; // remove this line if you're not using packages

/**
 * A short class to show how to test the Circle and Point classes.
 * This version contains a main method.
 */
public class CircleTester
{
   public CircleTester()
   {
   }

   /**
    * Test the Point and Circle classes.
    */
   public void doTest()
   {
      Point center = new Point(3,4);
      Circle c1 = new Circle();
      Circle c2 = new Circle(center, 5);
      Circle c3 = new Circle(3, 4, 5);
      System.out.println("c1 = " + c1);
      System.out.println("c2 = " + c2);
      System.out.println("c3 = " + c3);

      double radius = c2.getRadius();
      double x = c2.getCenter().getX();
      double y = c2.getCenter().getY();
      System.out.println("Radius = " + radius);
      System.out.println("Center x = " + x);
      System.out.println("Center y = " + y);
   }

   public static void main(String[] args)
   {
      CircleTester tester = new CircleTester();
      tester.doTest();
   }
}
```

This version of the class can be used both inside and outside BlueJ. The purpose of the `main` method here is to execute the statements we would do interactively with BlueJ: (1) an object called `tester` is constructed, (2) the `doTest` method is invoked on it. The `main` method can even be executed from within BlueJ. Just right click on the `CircleTester` rectangle and you will see `void main(args)` on the class menu. Select it and click OK to get them output as you would be creating an object with the constructor and invoking its `doTest` method.

Writing a runner class

The second approach which doesn't involve changing the class you want to run is to write a special runner class that has a `main` method in it. Example 4.21 shows a simple test of the `BankAccount` class using BeanShell. We can write this test as the following class:

Class `BankAccountRunner`

book-projects/chapter4/main_method

```
package chapter4.main_method; // remove this line if you're not using packages
import chapter4.bank_account.BankAccount; // remove this line if you're not using packages
/**
 * A class with only a main method showing how to do a simple test
 * of the BankAccount class outside BlueJ. The class is written so
 * that it can also be tested within BlueJ
 */
public class BankAccountRunner
{
   public void doTest()
   {
      BankAccount account = new BankAccount(123, "Peter Pascoe", 125.50);
      System.out.println("Initial balance is " + account.getBalance());
      account.withdraw(100);
      System.out.println("Balance is " + account.getBalance());
      account.deposit(100);
      System.out.println("Balance is " + account.getBalance());
      System.out.println(account);
   }
   public static void main(String[] args)
   {
      BankAccountRunner runner = new BankAccountRunner();
      runner.doTest();
   }
}
```

We have written this class so that it can also be run inside BlueJ in the usual way: (1) create a `BankAccountRunner` object called `runner` using the `new BankAccountRunner()` menu choice from the yellow class rectangle, (2) select the `doTest` choice from the object menu. The `main` method does exactly the same thing.

4.7 Running a class with a `main` method

To run a class with a `main` method outside BlueJ it is necessary to open a terminal window or console window (called an MS-DOS prompt or command prompt in windows) and then compile and interpret the class using the `javac` and `java` commands.

You can now navigate to the directory `c:/book-projects/chapter4/main_method` and issue the compiler command

```
javac BankAccountRunner.java
```

This command produces the bytecode file (see Chapter 1) called `BankAccountRunner.class`. You can use BlueJ do this step too and then open a command prompt and just use the interpreter.

To run the bytecode file use the interpreter command

```
java BankAccountRunner
```

The following output should appear in the terminal window.

```
Initial balance is 125.5
Balance is 25.5
Balance is 125.5
BankAccount[number=123, name=Peter Pascoe, balance=125.5]
```

4.8 Review exercises

► **Review Exercise 4.1** Define the following terms and give examples of each.

class implementation	class specification	public interface
unit testing	mutable class	immutable class
`Math.random`	`java.util.Random`	Java package
fully qualified name	`this`	association
aggregation	`toString`	`Date`
`Calendar`	`SimpleDateFormat`	`NumberFormat`
`DecimalFormat`	constructor call expression	factory method

default constructor	object reference	object reference variable
`null`	null reference	primitive type
reference type	object type	local variable
data encapsulation	data integrity	side-effects
copy constructor	instance variable	static variable
instance method	static method	method call expression
sending a message	method composition	static constant
formal argument	actual argument	call by value
`main` method	runner class	

► **Review Exercise 4.2** How can you easily distinguish a constructor prototype from a method prototype?

► **Review Exercise 4.3** Explain the relationship between a constructor declaration, a constructor prototype, and a constructor call expression.

► **Review Exercise 4.4** Explain the relationship between a method declaration, a method prototype, and a method call expression.

► **Review Exercise 4.5** Give three examples of a constructor prototype and for each example give three examples of a constructor call expression.

► **Review Exercise 4.6** Give three examples of a method prototype and for each example give three examples of a method call expression.

► **Review Exercise 4.7** If you do not provide a constructor in a class the compiler will provide a default constructor. What does this constructor do?

► **Review Exercise 4.8** What is the difference between a default constructor and a no-arg constructor?

► **Review Exercise 4.9** What is the difference between the public interface of a class and its implementation?

4.9 BeanShell exercises

► **BeanShell Exercise 4.1** Write some statements that define a string index, extract the character at that index, and display it.

► **BeanShell Exercise 4.2** Write some statements that define a string index, extract the character at that index as a one-character string, and display it.

► **BeanShell Exercise 4.3** Write some statements that define a start index and an end index, extract the substring whose first character is at the start index and whose last character is at the end index, and display the substring.

▶ **BeanShell Exercise 4.4** Write some statements that define two strings, search one string for the other string, and display the result.

▶ **BeanShell Exercise 4.5** Write some statements that convert a given string `s`, assumed to contain 4 lowercase letters, to a string called `alternate` of alternate upper and lower case letters. For example, if `s` has the value `"abcd"` then `alternate` has the value `"AbCd"`.

▶ **BeanShell Exercise 4.6** Define an all-uppercase string called `nameUpper` and write a single statement that converts it to a string called `name` in which the first letter is upper case and the remaining letters are lower case. For example, `"WILLIAM"` will be converted to `"William"`.

▶ **BeanShell Exercise 4.7** Write statements that construct two `BankAccount` objects each having an initial balance of $1,000. Use the `withdraw` and `deposit` methods to withdraw $50 from the first account and deposit it in the second account. Display the results.

▶ **BeanShell Exercise 4.8** Write statements that construct two `BankAccount` objects each having an initial balance of $1,000. Construct a `TransferAgent` object to transfer $50 from the first account to the second account. Display the results.

▶ **BeanShell Exercise 4.9** Write statements using `TriangleCalculatorTester` (page 118) to test the `TriangleCalculator` class.

▶ **BeanShell Exercise 4.10** Write some statements that use `Point` (page 119) to construct two points, calculate the distance between them, and display the results. (NOTE: it is necessary to use `import Point;` in BeanShell so that we get our `Point` class instead of `java.awt.Point`)

▶ **BeanShell Exercise 4.11** Write some statements that use `Circle` (page 122) to construct two circles and display the average of the `x` and `y` coordinates of their centers. (see Example 4.25)

▶ **BeanShell Exercise 4.12** Write some statements that use the `Calendar` class to determine what day of the week you were born on.

▶ **BeanShell Exercise 4.13** Write some statements that define a `double` number and use the `DecimalFormat` class to display this number in fixed format with 3 digits after the decimal point.

▶ **BeanShell Exercise 4.14** Write some statements that define a `double` number and use the `DecimalFormat` class to display this number in scientific format with 3 digits after the decimal point.

▶ **BeanShell Exercise 4.15** The `NumberFormat` class has a locale dependent static method called `getNumberInstance` that formats fixed point numbers for your locale. Experiment with the following statements:

```
NumberFormat fix5 = NumberFormat.getNumberInstance();
fix5.setMinimumFractionDigits(5);
fix5.setMaximumFractionDigits(5);
```

▶ **BeanShell Exercise 4.16** Write some statements to show the difference between call by value for primitive types and for a mutable reference type.

4.10 Programming exercises

In each programming exercise you should include javadoc comments and indicate what data you have used to test your class.

► **Exercise 4.1 (A `FullNameMaker` class)**
Write a class called `FullNameMaker` with the following interface.

```
public class FullNameMaker
{
   // data fields go here
   public FullNameMaker(String first, String mid, String last) {...}
   public String getName1() {...}
   public String getName2() {...}
   public String getName3() {...}
}
```

Here the three names of a person are provided to the constructor. Method `getName1` returns the full name, method `getName2` returns the full name but with the middle name replaced by its first letter followed by a period, and method `getName3` returns the last name first, followed by a comma and a space followed by the first name, followed by a space, the middle initial and a period. For example, if the three input names are ”WILLIAM”, ”James”, and ”duncan” then the methods should return the strings ”William James Duncan”, ”William J. Duncan”, and ”Duncan, William J.”. All names are stored and output with the first letter capitalized and other letters in lower case regardless of the input.

► **Exercise 4.2 (**A CopyCard class)
Make a project called `copy-card`. Using `BankAccount` as a model write a class called `CopyCard` that represents a student photocopy card using the student number, `studentID` (as an integer), the student name, `name` (as a `String`), and the amount remaining on the card, `balance` (as a `double` number). Include two methods that change the amount on the card: (1) an `add` method that adds a given amount to the copy card balance and (2) a `subtract` method that subtracts a given amount. The three “get” methods each return one of the data field values.

Also include in the `copy-card` project the following tester class that has a `doTest` method for use inside BlueJ and a `main` method for use outside BlueJ.

```
public class CopyCardTester
{
   public void doTest()
   {
      CopyCard fred = new CopyCard(123, "Fred Bolger", 125.0);
      CopyCard mary = new CopyCard(456, "Mary Nelson", 150.0);
      fred.subtract(25.0);
      mary.subtract(50.0);
      displayCard(fred);
      displayCard(mary);
      fred.add(20.0);
```

```
         mary.add(10.0);
         displayCard(fred);
         displayCard(mary);
      }

      private void displayCard(CopyCard c)
      {
         System.out.print("Id: " + c.getStudentID());
         System.out.print(", Name: " + c.getName());
         System.out.println(", Balance: " + c.getBalance());
      }

      public static void main(String[] args)
      {
         CopyCardTester tester = new CopyCardTester();
         tester.doTest();
      }
   }
```

Here we use a private method to display the results after each operation on the card. You can also run this class in BeanShell as follows. Make sure you select “Capture System in/out/err” from the BeanShell File menu and use `addClassPath` for your project. Then use the statements

```
CopyCardTester tester = new CopyCardTester();
tester.doTest();
```

► **Exercise 4.3 (A `Triangle` class)**
Make a project called `triangle` that contains the `Point` class from project
`book-projects/chapter4/geometry`
Add to this project the `Triangle` class with the public interface

```
public class Triangle
{
   public Point v1, v2, v3;
   public Triangle(Point p1, Point p2, Point p3) {...}
   public Point getVertex1() {...}
   public Point getVertex2() {...}
   public Point getVertex3() {...}
   public double area() {...}
   public String toString() {...}
}
```

that uses `Point` objects (aggregation) for the three vertices `v1`, `v2`, and `v3`. To calculate the area use the interesting formula

$$area = \frac{1}{2}\left|(x_2y_3 - x_3y_2) + (x_3y_1 - x_1y_3) + (x_1y_2 - x_2y_1)\right|$$

where (x_1, y_1), (x_2, y_2), and (x_3, y_3) are the three vertices. Recall that the absolute value of a number x is denoted by $|x|$. The method `Math.abs` can be used to calculate it.

► **Exercise 4.4 (A `Name` class)**
Make a BlueJ project called `person`. Write a class called `Name` that represents the name of a person using the following design.

```
public class Name
{
   private String firstName;
   private String lastName;

   public Name(String first, String last) {...}
   public Name(String fullName) {...}

   public String getFirstName() {...}
   public String getLastName() {...}
   public String toString() {...}
}
```

For the second constructor assume that a full name is the first name followed by one space followed by the last name. You can extract the two names using `indexOf` to find the position of the space.

Also, store the first and last names in a standard form. For example, ”james”, ”James”, and ”JAMES” should be stored as ”James”.

► **Exercise 4.5 (An `Address` class)**
Continuing Exercise 4.4, add to the `person` project an `Address` class that represents the address of a person using the following design.

```
public class Address
{
   String street;
   String city;
   String province;
   String postalCode;

   public Address(String street, String city, String province,
      String postalCode) {...}

   public String getStreet() {...}
   public String getCity() {...}
   public String getProvince() {...}
   public String getPostalCode() {...}
   public String toString() {...}
}
```

► **Exercise 4.6 (A simpler version of `Calendar`)**
Continuing Exercise 4.4 and Exercise 4.5, add to the `person` project an adapter class that is a simpler version of the complex `Calendar` class with the following public interface.

```
import java.util.Calendar;
/**
 * A simplified version of the complicated GregorianCalendar class that is
 * part of the standard Java library and contains about 50 methods. Our
 * class just needs to deal with the year, month, and the day. A class like
 * this is called an adapter class since it adapts a more complex class for
 * our simpler needs.
 */
public class CalendarDate
{
   private Calendar calendar;

   /**
    * Construct a calendar date for right now.
    */
   public CalendarDate() {...}

   /**
    * Construct a calendar date for a given year, month, and day.
    * @param year the year
    * @param month the month in the range 1 to 12
    * @param day the day of the month in range 1 to 31
    */
   public CalendarDate(int year, int month, int day) {...}

   /**
    * Return the year for this date.
    * @return the year for this date
    */
   public int getYear() {...}

   /**
    * Return the month for this date.
    * @return the month for this date in range 1 to 12
    */
   public int getMonth() {...}

   /**
    * Return the day of the month for this date.
    * @return the day of the month for this date in range 1 to 31
    */
   public int getDayOfMonth() {...}

   /**
    * Return a string representation of a calendar date.
    * @return a string representation of a calendar date
    */
   public String toString() {...}
}
```

▶ **Exercise 4.7 (A `Person` class using aggregation)**
Continuing Exercise 4.4, Exercise 4.5, and Exercise 4.6, add to the `person` project a `Person` class that represents the address of a person using the following design.

```
public class Person
{
   private Name name;
   private Address address;
   private CalendarDate birthDate;

   public Person(Name n, Address a) {...}

   public String getName() {...}
   public String getAddress() {...}
   public CalendarDate getBirthDate() {...}
   public int ageThisYear() {...}
   public String toString() {...}
}
```

▶ **Exercise 4.8 (New methods for the `Point` class)**
Make a project called `point` that contains the `Point` class from project
`book-projects/chapter4/geometry`
Include the following methods to the `Point` class.

(a) an instance method called `add` that takes a `Point` object as an argument and adds it to “this” `Point` object, returning a new `Point` object. Use the addition formula $(a,b)+(c,d) = (a+c,b+d)$.

(b) similarly include a `sub` method using the subtraction formula $(a,b)-(c,d) = (a-c,b-d)$.

(c) a static method called `add` that takes two `Point` objects as arguments, adds them, and returns a new `Point` object.

(d) similarly include a static `sub` method.

Chapter 5

Using Graphics Classes and Objects

An introduction to Java 2D

Outline

Drawing graphics in a frame

User space and device space in Java2D

Drawing process in Java2D

Writing classes using geometrical `Shape` objects

Using the `draw` and `fill` methods for shapes

Using `Point2D` to define points

`Line2D` Shape

`Rectangle2D` and `RoundRectangle2D` shapes

`Ellipse2D` and `Arc2D` shapes

Using `GeneralPath` to make custom shapes

Using attributes such as color and line thickness

Using affine transforms to do rotation, scaling and translation

Understanding coordinate transformations

5.1 Introduction

In this chapter we continue with the ideas presented in Chapter 4 but now we introduce some graphics classes and objects and learn how to use them to draw pictures. This will give you more experience using existing classes, writing simple classes, and computer graphics is fun too.

A graphical user interface (GUI) is necessary in order to draw pictures on a drawing surface. In Java we can do this in two ways: (1) with stand alone GUI applications, or (2) with applets that run in a browser. We will follow the first approach and write graphical applications that have a frame containing a rectangular drawing surface.

In order to make this as simple as possible for beginners we use a custom `GraphicsFrame` class. In Chapter 11 we explain how this class works, but for now we just use it to simplify the process of writing graphics programs that use Java 2D, the powerful two-dimensional graphics package that is part of Java. Java 2D is a large collection of classes and we will cover only the basic ideas in this book.

We begin with some important graphics concepts such as the graphics context, user space, and device space. Then we describe the part of the hierarchy of built-in geometrical shape classes that describes lines, rectangles, round rectangles, ellipses, and arcs. Some simple demonstration programs show how to draw these shapes.

Next we discuss the drawing process and how to specify attributes such as line thickness, colors, and rendering quality. The basic `draw` and `fill` methods are used to draw and fill shapes using the specified geometry and attributes. The entire process of specifying the geometry and attributes of objects, and then rendering the objects, is illustrated by several versions of a “happy face” program.

Finally, we illustrate how to change the coordinate system by calculating the transformations of points ourselves and by using the built-in `AffineTransform` class to change the default coordinate system. Affine transformations allow us to easily translate, scale, and rotate geometric objects before rendering them.

5.2 Using the `GraphicsFrame` class

The `GraphicsFrame` class is used to create a `JFrame` object representing a window on the screen, and to put a `JComponent` in it to act as a rectangular drawing surface or “canvas”, with a specified size, that exactly fills the interior of the frame.

This is a custom class that is available in `BlueJ` project `/book-projects/custom_classes` so you can just use “Add class from files” from the “Edit” menu to put a copy in each project that requires it.

For Windows an empty drawing window is shown in Figure 5.1. It has the usual title bar with the minimize, maximize and close buttons characteristic of a Windows program. The drawing surface is the interior of the frame, and is 400 pixels wide and 300 pixels high in this example.

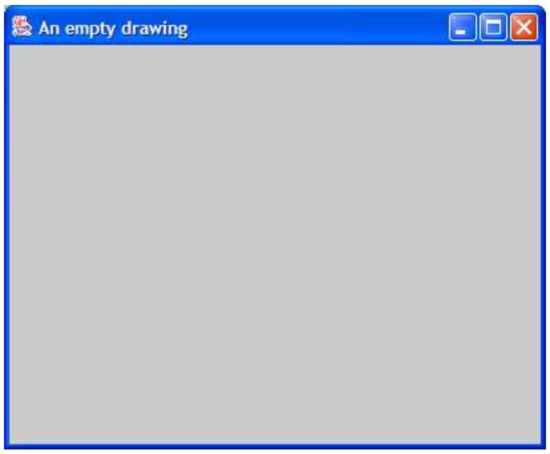


Figure 5.1: An empty drawing window

5.2.1 `EmptyDrawing` template for simple graphics programs

The empty frame in Figure 5.1 was produced by the following `EmptyDrawing` program class, which you can compile and run with the Java interpreter (`GraphicsFrame` should be in the same directory).

Class `EmptyDrawing`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A template for simple Java 2D drawing programs that use a
 * simple GraphicsFrame class that show a window frame with
 * a drawing surface (JPanel) inside.
 */
public class EmptyDrawing extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      // drawing statements go here
   }

   /**
    * Use this method inside BlueJ.
```

```
     */
    public void draw()
    {
       // Construct frame with a 400 pixel wide and 300 pixel high drawing area.
       // (0,0) is the top right corner and (399,299) is lower right corner.

       new GraphicsFrame("An empty drawing", new EmptyDrawing(), 400, 300);
    }

    /**
     * Use main method outside BlueJ from the command line.
     */
    public static void main(String[] args)
    {
       new EmptyDrawing().draw();
    }
}
```

We will use this class as a template for the programs in this chapter.

The first import statement is used only if you use the package version of the `GraphicsFrame` class. The other `import` statements are necessary to make available the standard Java classes for GUI programming. Since there are many classes that need to be imported we normally use the asterisk in the import statements to avoid a long list of import statements for the individual classes. The asterisk means to import all the classes in the package.

Next we have the name of our class. In the template this is `EmptyDrawing`. We also indicate that our class extends `JPanel`. The `JPanel` class defines a rectangular drawing surface, so the `EmptyDrawing` class extends this class. It will be our drawing canvas. It is an extension of a class called `JComponent`. We will learn in Chapter 10 that this extension mechanism, called inheritance and specified by the keyword `extends`, is a powerful way to organize related classes. For now just think of an `EmptyDrawing` object as a type of `JPanel` object which is also a type of `JComponent` object.

In a graphical Java user interface a **component** is an object that has a visible appearance on the screen (in Unix terminology components are often called widgets). The `paintComponent` method will contain our drawing statements as indicated by the comment in the `EmptyDrawing` class.

We write our classes so that they can be run both inside `BlueJ` using the `draw` method, and outside using the `main` method. In the `draw` method we simply construct the `GraphicsFrame` object

```
new GraphicsFrame("An empty drawing", new EmptyDrawing(), 400, 300);
```

We didn't need to give it a name so it is called an **anonymous** object. This corresponds to the custom `GraphicsFrame` constructor prototype (see Chapter 11)

```
public GraphicsFrame(String title, JComponent jc, int width, int height)
```

The first argument specifies a title for the window frame, `"An empty drawing"`, the next argument is an instance of our drawing class, `new EmptyDrawing()`, and the final two arguments specify the width, `400`, and height, `300`, of the drawing surface in pixels.

You can run the class in `BlueJ` as follows.

1. Right click on the `EmptyDrawing` rectangle and select `new EmptyDrawing`, corresponding to the default constructor
2. Right click on the resulting object and choose `void draw()` from the object menu.

To run the class outside BlueJ the `main` method just needs to construct an anonymous object of our class and call its `draw` method:

```
new EmptyDrawing().draw();
```

This is equivalent to

```
EmptyDrawing drawing = new EmptyDrawing();
drawing.draw();
```

which corresponds to the above Bluej actions.

5.3 The graphics context

In a windowing system graphics output is controlled by a **graphics context**. This context is a data structure which contains information on the type of graphics device (screen, printer, or plotter, for example), and how to render graphics on the device using the various drawing attributes, such as font information, line colors and styles, and region filling patterns and colors.

5.3.1 `Graphics` and `Graphics2D` objects

In Java a graphics context is an object from a class called `Graphics`. This class contains methods for drawing and setting attributes such as colors. In Java 1.1.x the support for graphics is very minimal; for example, there is no way to specify the thickness of lines in a drawing. When Java 1.2 (now called Java 2) was released, full support for two-dimensional graphics was provided by a rich set of classes referred to collectively as Java 2D.

To permit the use of the new graphics features, and also to maintain backward compatibility with the original ones, a subclass of the `Graphics` class called `Graphics2D` was provided. This is another example of inheritance. Since we will only be using Java 2D drawing methods, we need to convert an original graphics context `g` to a new one using the type cast statement

```
Graphics2D g2D = (Graphics2D) g;
```

This kind of type cast has the same syntax as the type casting of an `int` to a `double` using a statement such as "`double d = (int) i;`", with which we are familiar. However, it's purpose is quite different here and you will understand it only when we study inheritance in Chapter 10.

5.3.2 `paintComponent` method

To make pictures we need to put some statements in the `paintComponent` method. When a graphics program is run the `paintComponent` method is called by the window manager part of the Java run-time system. It is also called whenever part of the component needs to be refreshed. This can happen when part of the component is covered by another window and then uncovered, or when the component is restored after being minimized. We never directly call the `paintComponent` method ourselves. To draw a picture we only need to put our drawing statements in the body of this method, as indicated by the comment line in the `EmptyDrawing` class.

Also, through inheritance, a `JPanel` object is a `JComponent` object, so we need to call the `paintComponent` method in the parent `JComponent` class in case the component needs to be refreshed. This is done using the statement (see Chapter 10)

```
super.paintComponent(g);
```

which must always be the first statement in the `paintComponent` method body whose declaration for all program classes in this chapter will have the following structure:

```
public void paintComponent(Graphics g)
{
   super.paintComponent(g);
   Graphics2D g2D = (Graphics2D) g;

  // other statements go here
}
```

It is not necessary to understand the details at this stage. Just remember to include the two statements shown at the beginning of the method body.

Both the old and the new graphics contexts can be used. Methods in the original context can be referenced using `g`, and methods in the new one can be referenced using `g2D`. We never construct these graphics contexts ourselves. They are provided as actual arguments to the `paintComponent` method when the Java run-time system calls it to create the window or refresh it.

5.4 User space and device space

In order to draw lines, circles, rectangles and other objects we need to understand the coordinate systems used to specify the positions of objects on the drawing surface. All drawing commands use what is called user space. On the other hand, the rectangular drawing surface consists of discrete pixels (dots) arranged horizontally and vertically in what is called a device space. The coordinate system in **device space** is an integer coordinate system and each pixel is represented by its integer coordinates x and y. Every graphics device (screen, printer, or plotter, for example) has a device space associated with it. Since we will only be using the device space for the rectangular screen window, we often refer to device space as **screen space**.

For screen space, the origin $(0,0)$ is at the top left corner of the surface. The horizontal x coordinate increases from left to right and the vertical y coordinate increases from top to bottom. Unfortunately, this is a left-handed coordinate system, instead of the universal right-handed coordinate system used in mathematics, for which the origin $(0,0)$ is at the bottom left and the y axis increases from bottom to top. Later we will see how to change the coordinate system to the conventional one if desired.

User space is a space of double precision numbers that represents a continuous space. By default, the ranges of its coordinates are the same as that of the screen space. For example, if we have a 31 by 21 pixel drawing surface then the user origin $(0.0, 0.0)$ corresponds to the top left pixel, which has integer screen coordinates $(0, 0)$, and the bottom right corner would be $(30.0, 20.0)$, corresponding to the pixel with screen coordinates $(30, 20)$. This is shown in Figure 5.2.

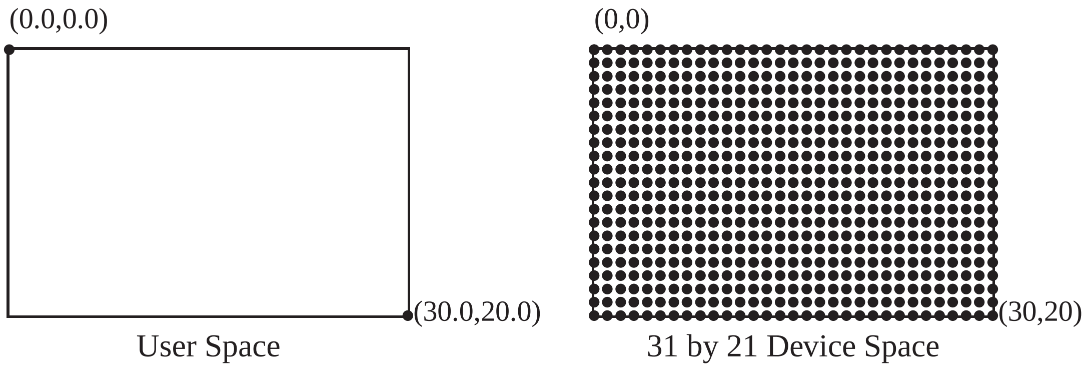


Figure 5.2: User and device space

It is important to realize that the width and height in device space are not measured in some system of units. They are the width and height in pixels. In Figure 5.2 device space is 31 pixels wide and 21 pixels high but the coordinates of the lower right corner are 30 and 20. This does not apply to user space; here the width is 30 units measured using the double precision coordinate system.

In the general case a w by h pixel device space has top left corner at $(0, 0)$ and bottom right corner at $(w-1, h-1)$ and corresponds to a user space with top left corner at (0.0,0.0) and bottom right corner at $(w-1, h-1)$. The specific transformation from user space to device space is called the **default transformation**. It is part of the graphics context and is automatically provided for us each time the `paintComponent` method is called. Later we will see how to modify the default transformation.

5.5 Graphics classes and objects

In order to make drawings we need some classes that describe the geometry of basic graphical objects, such as lines, rectangles and circles, in terms of points in user space. A rich set of classes is provided with Java2D.

5.5.1 `Point2D` and `Line2D` classes

The simplest classes, `Point2D` and `Line2D`, in the `java.awt.geom` package, are used to define points and lines.

Defining points

Points in user space can either be represented by their *x* and *y* coordinates, or as `Point2D` objects. For example, to define the four corner points in user space in Figure 5.2 we could define the 4 objects

```
Point2D.Double topLeft = new Point2D.Double(0.0, 0.0);
Point2D.Double topRight = new Point2D.Double(30.0, 0.0);
Point2D.Double bottomLeft = new Point2D.Double(0.0, 20.0);
Point2D.Double bottomRight = new Point2D.Double(30.0, 20.0);
```

The class name here has the qualified name `Point2D.Double` indicating that points are represented by double precision numbers. There is also a class called `Point2D.Float` that can represent points using single precision floating point numbers. Then we would use the statements

```
Point2D.Float topLeft = new Point2D.Float(0.0F, 0.0F);
Point2D.Float topRight = new Point2D.Float(30.0F, 0.0F);
Point2D.Float bottomLeft = new Point2D.Float(0.0F, 20.0F);
Point2D.Float bottomRight = new Point2D.Float(30.0F, 20.0F);
```

Here it is necessary to tell the compiler that the decimal constants are of type `float` rather than `double` using the `F` suffix, since the default is `double` if no suffix is used. The compiler does not implicitly convert values of type `double` to `float` and gives an "incompatible type" error if the `F` suffix is omitted. The prototypes for these constructors are

```
public Point2D.Float(float x, float y)
public Point2D.Double(double x, double y)
```

After a point has been constructed, the individual coordinates can be retrieved using the `getX` and `getY` methods which have the prototypes

```
public double getX()
public double getY()
```

These enquiry methods return `double` values even for points of type `Point2D.Float`.

Defining lines using points

Lines in user space can be described geometrically as objects from the `Line2D` class using their end points. There are two classes: `Line2D.Double` and `Line2D.Float`. The constructor prototypes are

```
public Line2D.Float(Point2D.Float p1, Point2D.Float p2)
public Line2D.Double(Point2D.Double p1, Point2D.Double p2)
```

where `p1` and `p2` are the end points of the lines.

For example, to define a line between the points `topLeft` and `topRight` and draw it we could use the statements.

```
Line2D.Double line = new Line2D.Double(topLeft, topRight);
g2D.draw(line);
```

which uses the constructor that takes two points as arguments. Defining a line object does not cause the line to be drawn. It only describes the geometry. In order to draw the line we must use the `draw` method in the graphics context. If we don't need to refer to the line again, we can write these statements as the single statement

```
g2D.draw(new Line2D.Double(topLeft, topRight));
```

which uses an anonymous object.

Here is a simple program class, containing a `main` method, that draws a line and labels the two endpoints with their coordinates:

Class `DrawLine`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Draw a line and label its endpoints with their coordinates.
 */
public class DrawLine extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      Point2D.Double p1 = new Point2D.Double(10.0,20.0);
      Point2D.Double p2 = new Point2D.Double(225.0,170.0);
      g2D.draw(new Line2D.Double(p1,p2));
      g2D.drawString("(10,20)", (float) p1.getX(), (float) p1.getY());
      g2D.drawString("(225,170)", (float) p2.getX(), (float) p2.getY());
   }

   public void draw()
   {
      new GraphicsFrame("Drawing a line", new DrawLine(), 300, 200);
   }

   public static void main(String[] args)
   {
      new DrawLine().draw();
   }
}
```

The `drawString` method is also illustrated here. It is the simplest way to draw text. There are two method prototypes

```
public void drawString(String text, int x, int y)
public void drawString(String text, float x, float y)
```

where (x,y) are the coordinates of the bottom left corner of the text string. Since there is no version that takes `double` arguments, we needed to type cast the coordinates returned by `getX()` and `getY()` to type `float` in the `DrawLine` class. The output is shown in Figure 5.3.

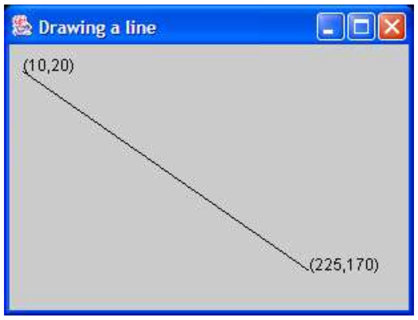


Figure 5.3: Drawing a line

Drawing rectangles using lines

As another simple example, here is a program that draws the biggest rectangle on the drawing surface inside the frame:

Class `DrawBiggestRectangle`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Draw the biggest rectangle inside the frame using lines to draw the
 * rectangle. If the frame is resized the rectangle doesn’t change.
*/
public class DrawBiggestRectangle extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      Point2D.Double topLeft = new Point2D.Double(0.0, 0.0);
      Point2D.Double topRight = new Point2D.Double(300.0, 0.0);
      Point2D.Double bottomLeft = new Point2D.Double(0.0, 200.0);
      Point2D.Double bottomRight = new Point2D.Double(300.0, 200.0);

      Line2D.Double top = new Line2D.Double(topLeft, topRight);
      Line2D.Double right = new Line2D.Double(topRight, bottomRight);
      Line2D.Double bottom = new Line2D.Double(bottomRight, bottomLeft);
      Line2D.Double left = new Line2D.Double(bottomLeft, topLeft);

      g2D.draw(top);
      g2D.draw(right);
      g2D.draw(bottom);
      g2D.draw(left);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing the biggest rectangle",
                        new DrawBiggestRectangle(), 301, 201);
   }

   public static void main(String[] args)
   {
      new DrawBiggestRectangle().draw();
   }
}
```

The program output is shown in Figure 5.4. The black rectangle outline is just visible at the border

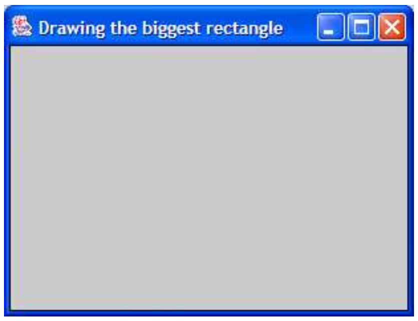


Figure 5.4: The biggest rectangle

of the drawing surface.

Defining lines using coordinates

The `Line2D` classes have another constructor that uses 4 numbers to specify the coordinates of the endpoints. They have the prototypes

```
public Line2D.Float(float x1, float y1, float x2, float y2)
public Line2D.Double(double x1, double y1, double x2, double y2)
```

where `x1` and `y1` are the coordinates of one end point and `x2` and `y2` are the coordinates of the other end point. In the `DrawBiggestRectangle` program we can replace the eight statements defining the points and lines by the four statements

```
Line2D.Double top = new Line2D.Double(0.0, 0.0, 300.0, 0.0);
Line2D.Double right = new Line2D.Double(300.0, 0.0, 300.0, 200.0);
Line2D.Double bottom = new Line2D.Double(300.0, 200.0, 0.0, 200.0);
Line2D.Double left = new Line2D.Double(0.0, 200.0, 0.0, 0.0);
```

Resizing the drawing window

When the window is displayed, it can be resized by clicking on any of the sides or corners and dragging the mouse. Figure 5.5 shows the `DrawBiggestRectangle` window after resizing. The

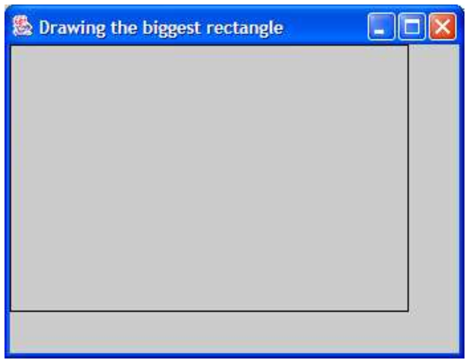


Figure 5.5: The biggest rectangle after resizing window

rectangle did not automatically expand to become the biggest one possible in the new window. This may or may not be what you want. If you want the rectangle to resize itself so that it is always the biggest one, then it is necessary to ask the graphics context for the current size of the drawing surface. We can do this by adding the following two statements to the `paintComponent` method.

```
double xMax = getWidth() - 1;
double yMax = getHeight() - 1;
```

The `getWidth` and `getHeight` methods return the width and height of the drawing surface in pixels. We subtract 1 from each to get the integer coordinates of the bottom right pixel. The right sides of these assignments evaluate to integer values. These values are automatically converted to double values in user space (explicit type conversion) for assignment to `xMax` and `yMax`.

The instance methods `getWidth` and `getHeight` don't need to use the dot notation to send a message to an object. When a method is used in this way it is implied that it is sending the message to an object of the class in which the method is used. This object is actually a `JComponent` object, and if you were to look at this class you would find that it has a `getWidth` method and a `getHeight` method.

Now change the definitions of the four corner points in the `DrawBiggestRectangle` program to

```
Point2D.Double topLeft = new Point2D.Double(0.0, 0.0);
Point2D.Double topRight = new Point2D.Double(xMax, 0.0);
Point2D.Double bottomLeft = new Point2D.Double(0.0, yMax);
Point2D.Double bottomRight = new Point2D.Double(xMax, yMax);
```

to obtain the following class.

Class `DrawBiggestRectangle2`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Another version of DrawBiggestRectangle that uses the component’s
 * getWidth and getHeight methods to draw the biggest rectangle even
 * when the frame is resized.
 */
public class DrawBiggestRectangle2 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      double xMax = getWidth() - 1;
      double yMax = getHeight() - 1;

      Point2D.Double topLeft = new Point2D.Double(0.0, 0.0);
      Point2D.Double topRight = new Point2D.Double(xMax, 0.0);
      Point2D.Double bottomLeft = new Point2D.Double(0.0, yMax);
      Point2D.Double bottomRight = new Point2D.Double(xMax, yMax);

      Line2D.Double top = new Line2D.Double(topLeft, topRight);
      Line2D.Double right = new Line2D.Double(topRight, bottomRight);
      Line2D.Double bottom = new Line2D.Double(bottomRight, bottomLeft);
      Line2D.Double left = new Line2D.Double(bottomLeft, topLeft);

      g2D.draw(top);
      g2D.draw(right);
      g2D.draw(bottom);
      g2D.draw(left);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing the biggest rectangle (2)",
                        new DrawBiggestRectangle2(), 301, 201);
   }

   public static void main(String[] args)
   {
      new DrawBiggestRectangle2().draw();
   }
}
```

When you run this class the rectangle will be resized automatically when the window is resized. This works only because `getWidth` and `getHeight` are inside the `paintComponent` method. Each time the window is resized this method is called and the variables `xMax` and `yMax` will have new values to reflect the new size of the drawing surface.

5.5.2 Geometrical Shape Hierarchy (`java.awt.geom` package)

The `Line2D` class is one of an extensive set of graphical classes. Each class represents a geometrical object as a `Shape` as shown in Figure 5.6. We will study the concept of inheritance and interfaces in Chapter 10. All the objects can be considered as different kinds of `Shape` objects. They belong to the `Shape` family. The nice thing about the `Shape` family is that the `draw` method in the `Graphics2D` graphics context knows how to draw objects defined by any of the classes in this family. In fact it has the prototype

```
public void draw(Shape s)
```

This is much better than having separate methods such as `drawLine`, `drawRectangle`, or `drawArc` to draw each kind of object. The two `Point2D` classes are also shown but they are not part of the `Shape` family.

Drawing points

You may wonder how to draw a single point. If `p` is a `Point2D` object, then you might expect that `g2D.draw(p)` will draw the point. This does not work since a point is not a `Shape`. Instead it is necessary to draw a line from one point to the same point using

```
g2D.draw(new Line2D.Double(p,p));
```

Using the coordinates *x* and *y* of the point you can also use

```
g2D.draw(new Line2D.Double(x,y,x,y));
```

If you want to draw pictures entirely in terms of pixels in device space rather than user space there are better ways, which we will not discuss, that use image and raster objects.

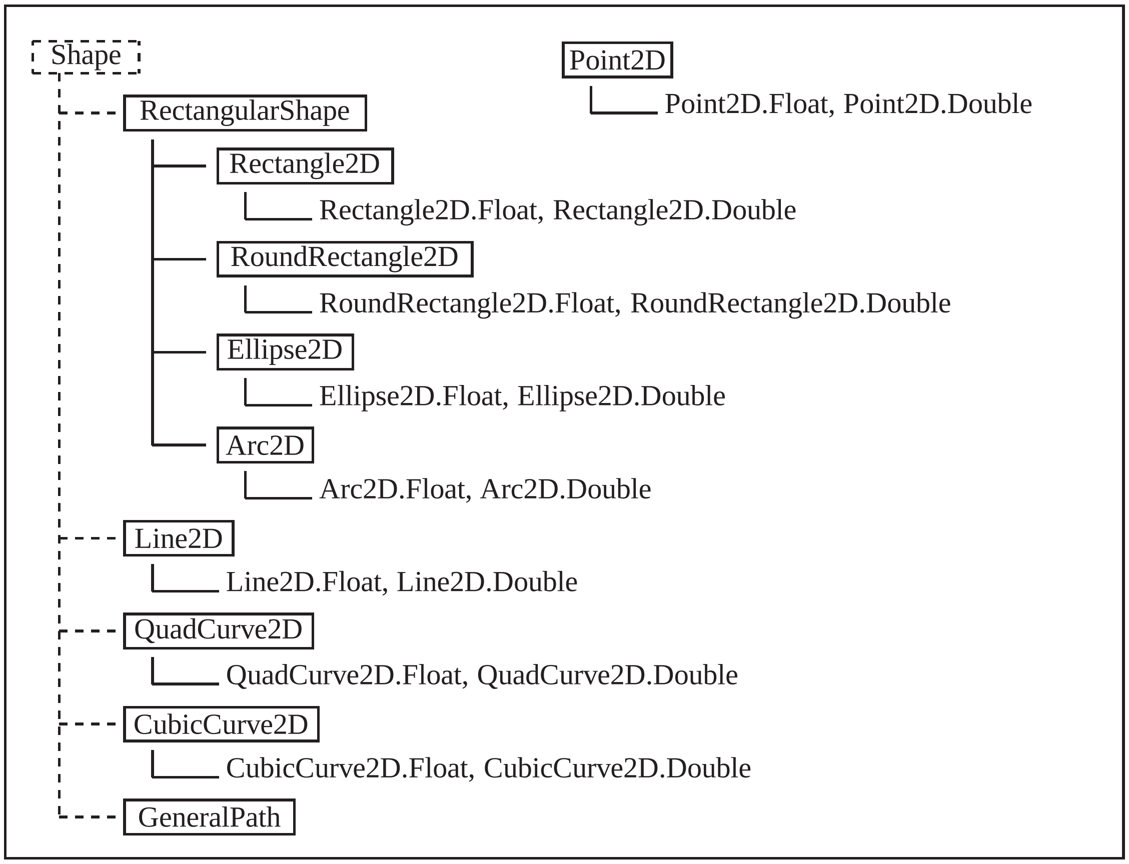

Shape
RectangularShape
Rectangle2D
Rectangle2D.Float, Rectangle2D.Double
RoundRectangle2D
RoundRectangle2D.Float, RoundRectangle2D.Double
Ellipse2D
Ellipse2D.Float, Ellipse2D.Double
Arc2D
Arc2D.Float, Arc2D.Double
Line2D
Line2D.Float, Line2D.Double
QuadCurve2D
QuadCurve2D.Float, QuadCurve2D.Double
CubicCurve2D
CubicCurve2D.Float, CubicCurve2D.Double
GeneralPath
Point2D
Point2D.Float, Point2D.Double

Figure 5.6: Inheritance hierarchy and `Shape` interface diagram for graphics classes

5.5.3 `RectangularShape` classes

The four classes of type `RectangularShape` are defined in terms of the frame (a rectangle) of the shape. Do not confuse the use of the word frame here with the window frame. There is also the concept of the bounding rectangle (or bounding box) for a shape. It is the smallest rectangle containing the shape and is not always the same rectangle as the frame.

- A `Rectangle2D` object defines a rectangle using its frame. Here the frame is the same as the bounding rectangle.

- An `Ellipse2D` object defines an ellipse using its frame. Here the frame is the same as the bounding rectangle.

- A `RoundRectangle2D` object defines a rounded rectangle that has elliptical arcs at the corners. It is defined using its frame and the frame of the ellipse that contains the corner arcs.

- An `Arc2D` object defines an arc of an ellipse using its frame. For an arc the frame and the bounding rectangle are different rectangles.

`Rectangle2D` class

We have defined a rectangle using four lines in `DrawBiggestRectangle` but the `Rectangle2D` class is more convenient. The prototypes for constructing a rectangle object are

```
public Rectangle2D.Float(float x, float y, float w, float h)
public Rectangle2D.Double(double x, double y, double w, double h)
```

where `x` and `y` are the coordinates of the top left corner of the rectangle, `w` is the width of the rectangle, and `h` is the height of the rectangle.

In the `DrawBiggestRectangle2` class we can replace the `paintComponent` method to obtain the class

Class `DrawBiggestRectangle3`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A version of DrawBiggestRectangle2 that uses a Rectangle2D.Double
 * object instead of four Line2D.Double objects to draw the biggest rectangle.
 */
public class DrawBiggestRectangle3 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      double xMax = getWidth() - 1;
      double yMax = getHeight() - 1;

      Rectangle2D.Double biggestRect =
         new Rectangle2D.Double(0.0, 0.0, xMax, yMax);

      g2D.draw(biggestRect);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing the biggest rectangle (3)",
                        new DrawBiggestRectangle3(), 301, 201);
   }

   public static void main(String[] args)
   {
      new DrawBiggestRectangle3().draw();
   }
}
```

which defines a rectangle in user space with top left corner at `(0.0,0.0)`, width `xMax - 0.0 = xMax` and height `yMax - 0.0 = yMax`.

`Ellipse2D` class

The `Ellipse2D.Float` and `Ellipse2D.Double` classes define ellipses. The constructors are

```
public Ellipse2D.Float(float x, float y, float w, float h)
public Ellipse2D.Double(double x, double y, double w, double h)
```

Here `x` and `y` are the coordinates of the top left corner of the ellipse frame, `w` is the width of this frame, and `h` is its height. The ellipse will be a circle if `w` and `h` are the same. Here is a class that draws an ellipse and a circle with their frames.

Class `DrawEllipse`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Show how to draw an ellipse and a circle and their bounding boxes.
 */
public class DrawEllipse extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      Ellipse2D.Double ellipse = new Ellipse2D.Double(20,20,200,100);
      Ellipse2D.Double circle = new Ellipse2D.Double(300,20,100,100);

      g2D.draw(ellipse);
      g2D.draw(ellipse.getFrame());
      g2D.draw(circle);
      g2D.draw(circle.getFrame());
   }

   public void draw()
   {
      new GraphicsFrame("Drawing ellipses", new DrawEllipse(), 422, 142);
   }
   public static void main(String[] args)
   {
      new DrawEllipse().draw();
   }
}
```

The first ellipse constructor defines an ellipse whose frame has top left corner at (20,20) with width 200 and height 100. The second ellipse constructor defines a circle whose frame has top left corner at (300,20) with width and height 100. This gives a circle of radius 50. The output is shown in Figure 5.7. The left window shows the ellipse and circle with their frames and the right window,

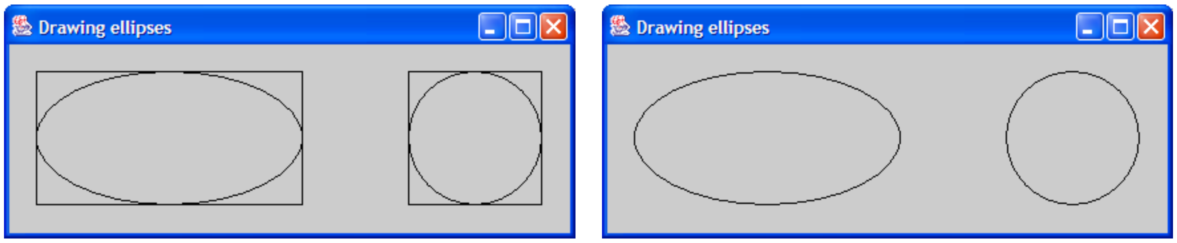


Figure 5.7: Ellipse, circle, and frames

obtained by commenting out the statements in the class that define and draw the rectangles, shows the ellipse and circle without their frames. There is also a `getBounds2D` method which will return the bounding box. In the case of an ellipse, `GetBounds2D` returns the same rectangle as `getFrame`.

`RoundRectangle2D` class

The `RoundRectangle2D.Float` and `RoundRectangle2D.Double` classes define rectangles with rounded corners. The corners are one quarter arcs of an ellipse. The constructor prototypes are

```
public RoundRectangle2D.Float(float x, float y, float w, float h,
   float arcw, float arch)

public RoundRectangle2D.Double(double x, double y, double w, double h
   double arcw, double arch)
```

Here `x` and `y` are the coordinates of the top left corner of the frame, `w` is its width, and `h` is its height. The values of `arcw` and `arch` define the width and height of the frame of the ellipse of which the arc is a part. Here is a program that draws two round rectangles of the same size but with different sized arcs at the corners.

Class `DrawRoundRectangle`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Draw rounded rectangles using RoundRectangle2D
 */
public class DrawRoundRectangle extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      // First four parameters define the frame and last
      // two parameters define the arc width and height

      RoundRectangle2D.Double roundRect1 =
         new RoundRectangle2D.Double(20,20,200,100,100,60);
      RoundRectangle2D.Double roundRect2 =
         new RoundRectangle2D.Double(240,20,200,100,100,100);

      g2D.draw(roundRect1);
      g2D.draw(roundRect2);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing round rectangles",
         new DrawRoundRectangle(), 462, 142);
   }

   public static void main(String[] args)
   {
      new DrawRoundRectangle().draw();
   }
}
```

The round rectangle on the left has top left corner at (20,20), width 200, and height 100. It has an arc of an ellipse whose frame has width 100 and height 60. This means that the arc is 50 units wide and 30 units tall. The round rectangle on the right has the same size with top left corner at (240,20) but it has a circular arc that is 50 units wide and 50 units high. Since this is half the height of the round rectangle we obtain a round rectangle with semi-circular sides. The output is shown in Figure 5.8.

`Arc2D` class

The `Arc2D.Float` and `Arc2D.Double` classes define an arc as a part of an ellipse. The constructor prototypes are

```
      public Arc2D.Float(float x, float y, float w, float h,
```

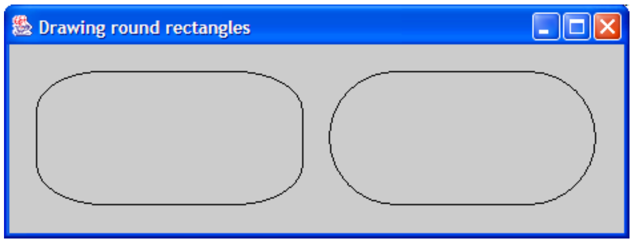


Figure 5.8: Drawing round rectangles

```
        float startAngle, float extentAngle, int type)

     public Arc2D.Double(double x, double y, double w, double h,
        double startAngle, double extentAngle, int type)
```

As with the other `RectangleShape` classes, the first four arguments define the frame of the ellipse of which the arc is a part. The value of `startAngle` specifies the starting angle in degrees for the arc and `extentAngle` specifies how many degrees are in the arc. A horizontal line to the right of the arc center has an angle of 0. The `type` argument has one of the three constant values `Arc2D.OPEN` for an ordinary arc, `Arc2D.CHORD` if the start and end points of the arc should be joined, and `Arc2D.PIE` if the end points of the arc should be joined to the center of the ellipse. Here is a class that draws an arc using the three different joining conditions:

Class `DrawArc`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Show how to draw arcs using the three types of joining conditions.
 */
public class DrawArc extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      // First four parameters give the frame of enclosing ellipse

      Arc2D.Double arc1 = new Arc2D.Double(20,20,200,100,0,210,Arc2D.OPEN);
      Arc2D.Double arc2 = new Arc2D.Double(240,20,200,100,0,210,Arc2D.CHORD);
      Arc2D.Double arc3 = new Arc2D.Double(460,20,200,100, 0, 210,Arc2D.PIE);
```

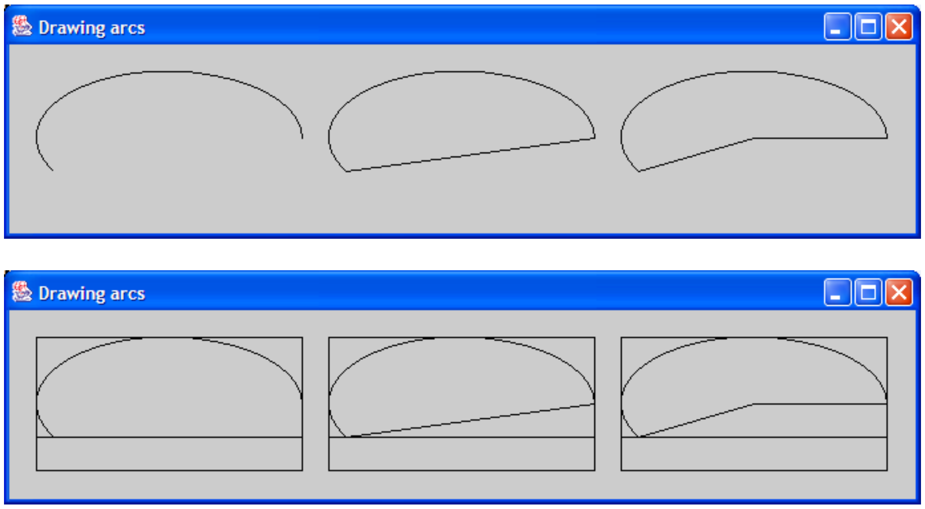


Figure 5.9: The three types of arcs

```
        /* For arcs the frame is not the same as the bounding box. The Frame
         * is the frame of the ellipse of which the arc is a part and the
         * bounding box is the smallest rectangle enclosing the arc
         */

        g2D.draw(arc1);
        g2D.draw(arc1.getFrame());
        g2D.draw(arc1.getBounds2D());
        g2D.draw(arc2);
        g2D.draw(arc2.getFrame());
        g2D.draw(arc2.getBounds2D());
        g2D.draw(arc3);
        g2D.draw(arc3.getFrame());
        g2D.draw(arc3.getBounds2D());
    }

    public void draw()
    {
        new GraphicsFrame("Drawing arcs", new DrawArc(), 682, 142);
    }

    public static void main(String[] args)
    {
        new DrawArc().draw();
    }
}
```

The first arc has a frame with top left corner at (20,20), width 200, and height 100. The arc extends from angle 0 (horizontal) in a counterclockwise direction 210 degrees. The other arcs have the same angles. The output is shown in Figure 5.9. The bottom window shows the three arcs with their frames and the top window shows them without frames. As mentioned above, for arcs `getFrame` and `getBounds2D` return different rectangles.

5.6 The drawing process

So far our programs have had the following structure in the `paintComponent` method: define the geometry, then render it with the draw command. There are actually four steps to the drawing process:

1. Define the geometry of the picture in some convenient coordinate system.
2. Transform the geometry if necessary to position objects on the drawing surface.
3. Specify any attributes such as colors or line thicknesses.
4. Render the drawing using draw and fill commands and attributes.

We have used the default user space coordinate system, which is just the device space coordinate system but using double numbers instead of integers. In steps 2 and 3 it is possible to define any suitable coordinate system and to specify attributes such as drawing quality, colors, line styles, and line thickness.

5.6.1 Specifying attributes

The `draw` command uses a default set of attributes to render a geometrical object. For example, the default color is black, the default screen background is light gray, the line thickness is one pixel, and the rendering of curves and lines is not anti-aliased.

Improving the rendering quality (smoothness)

When lines and curves are drawn they often look jagged. The effect is more noticeable the nearer the line is to the horizontal or vertical. To a certain extent this is an unavoidable consequence of using discrete pixels to represent a continuous mathematical line. The effect can be minimized by using what is called “anti-aliasing”. With this technique, pixels near the ones that are drawn can be given various shades of gray or other colors to make the lines appear smoother.

In Java 2 it is possible to turn on anti-aliasing using the following formidable looking statement

```
g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
```

where `g2D` is the graphics context. Place this statement in the `paintComponent` method after the definition of `g2D` in any of the classes we have considered and you will notice a dramatic increase in picture quality. For example if the `paintComponent` method of the `DrawArc` program is replaced by

```
public void paintComponent(Graphics g)
{
   super.paintComponent(g);
   Graphics2D g2D = (Graphics2D) g;

   g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

   Arc2D.Double arc1 = new Arc2D.Double(10,10,100,50,0,210,Arc2D.OPEN);
   Arc2D.Double arc2 = new Arc2D.Double(120,10,100,50,0,210,Arc2D.CHORD);
   Arc2D.Double arc3 = new Arc2D.Double(230,10,100,50,0,210,Arc2D.PIE);
   g2D.draw(arc1);
   g2D.draw(arc2);
   g2D.draw(arc3);
}
```

then the result shown in Figure 5.10 can be compared with the top frame in Figure 5.9.

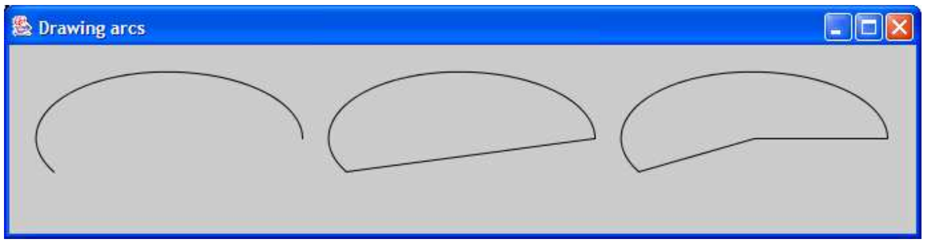


Figure 5.10: Anti-aliased version of `DrawArc`

Specifying colors

Colors in Java 2D are objects from a class called `Color` in package `java.awt`. There are several models for dealing with color in graphics systems. We consider only the RGB system in which colors are specified by their red, green, and blue components. The two constructors have prototypes

```
public Color(int red, int green, int blue)
public Color(float red, float green, float blue)
```

With the integer arguments, color values are in the range 0 to 255 with 0 being the absence of the color. In the floating point version the values are in the range `0.0F` to `1.0F`.

■ EXAMPLE 5.1 **(Defining colors)** The statements

```
Color redColor = new Color(255,0,0);
Color greenColor = new Color(0,255,0);
Color blueColor = new Color(0,0,255);
Color blackColor = new Color(0,0,0);       // absence of all colors
Color whiteColor = new Color(255,255,255); // presence of all colors
Color yellowColor = new Color(255,255,0);  // red + green = yellow
```

Name	**Red**	**Green**	**Blue**	**Name**	**Red**	**Green**	**Blue**
`Color.black`	0	0	0	`Color.blue`	0	0	255
`Color.cyan`	0	255	255	`Color.darkGray`	64	64	64
`Color.gray`	128	128	128	`Color.green`	0	255	0
`Color.lightGray`	192	192	192	`Color.magenta`	255	0	255
`Color.orange`	255	200	0	`Color.pink`	255	175	175
`Color.red`	255	0	0	`Color.white`	255	255	255
`Color.yellow`	255	255	0				

Table 5.1: The standard colors

define some standard colors using the integer arguments. ■

There are 13 predefined color constants in class `Color` which can also be used. Their names and values are shown in Table 5.1. There are two useful methods in the `Graphics2D` graphics context for specifying colors. The `setPaint` method is used to set the drawing color, and the `getPaint` method is used to retrieve the current drawing color in case you want to save it. They have the prototypes

```
public void setPaint(Paint p)
public Paint getPaint()
```

A `Color` object is a kind of `Paint` object so the statement

```
g2D.setPaint(Color.red);
```

sets the current drawing color to red. All subsequent `draw` commands will use this color until it is changed. The statement

```
Paint save = g2D.getPaint();
```

saves the current color in variable `save`.

■ EXAMPLE 5.2 **(Colored arcs)** In the `paintComponent` method for the `DrawArc` program given on page 200 you can replace the three draw statements for the arcs by

```
g2D.setPaint(Color.red);
g2D.draw(arc1);
g2D.setPaint(Color.green);
g2D.draw(arc2);
g2D.setPaint(Color.blue);
g2D.draw(arc3);
```

to get a red, green, and a blue arc. ■

Specifying line thickness

Another important attribute in the rendering of a graphics object is the line thickness. When a line or curve is rendered it is stroked by a brush with a certain thickness. This brush is an object from a class called `Stroke` which has a useful subclass called `BasicStroke`. One of the constructors has the prototype

```
public BasicStroke(float width)
```

where `width` is the thickness of the brush in user space. The `setStroke` method in the graphics context is used to make a `Stroke` object the current one. The default thickness is 1 pixel. There are other constructors, which we won't use in this book, that specify how lines are joined.

■ EXAMPLE 5.3 **(Specifying line thickness)** In the `DrawArc` program on page 200 if we add the statement

```
g2D.setStroke(new BasicStroke(2.0F));
```

before the `draw` statements, the arcs will be drawn with curves and lines that are twice as thick as the default (obtained using `1.0F`). ■

Here is the final version of the arc drawing program called `DrawArc2` that draws three arcs in red, green, and blue, with anti-aliasing and a stroke thickness of 2:

Class `DrawArc2`

book-projects/chapter5/simple_shapes

```
package chapter5.simple_shapes; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A version of DrawArc that uses colors, line thickness,
 * and anti-aliasing for a smoother picture.
 */
public class DrawArc2 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      Arc2D.Double arc1 = new Arc2D.Double(20,20,200,100,0,210,Arc2D.OPEN);
      Arc2D.Double arc2 = new Arc2D.Double(240,20,200,100,0,210,Arc2D.CHORD);
      Arc2D.Double arc3 = new Arc2D.Double(460,20,200,100,0,210,Arc2D.PIE);
      g2D.setStroke(new BasicStroke(2.0f)); // lines twice as wide as default
      g2D.setPaint(Color.red);
      g2D.draw(arc1);
      g2D.setPaint(Color.green);
      g2D.draw(arc2);
      g2D.setPaint(Color.blue);
      g2D.draw(arc3);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing arcs (2)", new DrawArc2(), 682, 142);
   }

   public static void main(String[] args)
   {
      new DrawArc2().draw();
   }
}
```

The output window is shown in Figure 5.11.

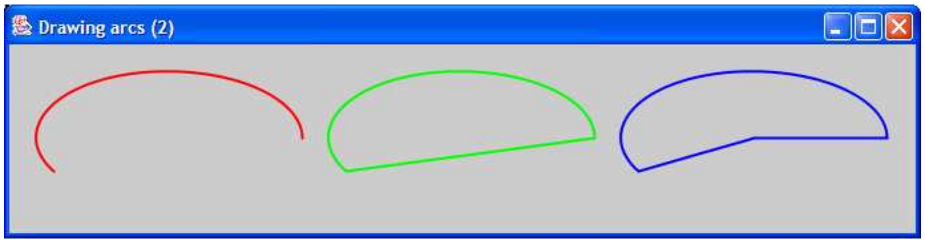


Figure 5.11: `DrawArc2` showing colors and stroke thickness 2

Filling and stroking

Another important operation is to fill the interior of an object with a color. For this we use the `fill` command with prototype

```
public void fill(Shape s)
```

■ EXAMPLE 5.4 **(Filling and stroking)** The statements

```
Ellipse2D.Double circle = new Ellipse2D.Double(40,40,120,120);
g2D.setPaint(Color.red);
g2D.fill(circle);
g2D.setPaint(Color.blue);
g2D.setStroke(new BasicStroke(2.0f));
g2D.draw(circle);
```

fill a circle of radius 60 centered at (100,100) with red and then stroke its outline in blue using a stroke of width 2 in user space. ■

5.7 Put on a happy face

Let us put everything we have learned together to write some classes that draw a happy face with eyes, a nose, and a mouth. First we design the face using boxes. Then we draw a no-frills version of the face. Next we use filling and colors. Finally we show how our face can be transformed by rotations, scalings, and translations to produce different faces, all from the original version.

5.7.1 Designing the face with boxes

We can use a circle for the face, two circles for the eyes, a triangle for the nose, and the bottom half of an ellipse for the mouth. Since circles, arcs, and ellipses are specified using their frames it is easy to first lay out the boxes as shown in Figure 5.12. Here we are using a drawing surface

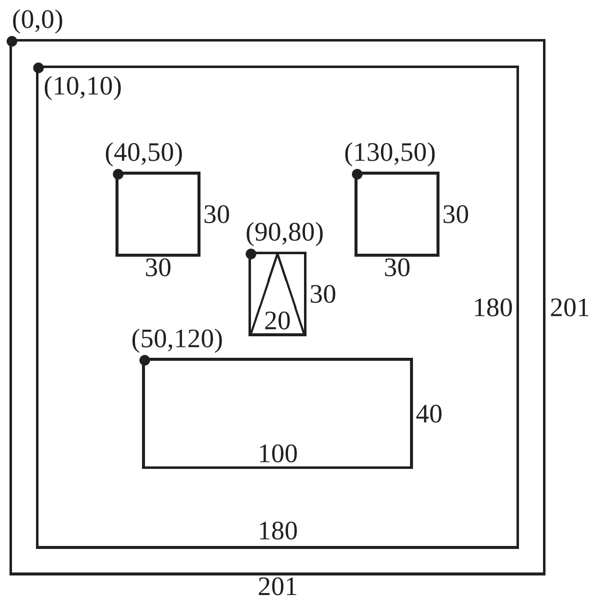


Figure 5.12: Designing the happy face with boxes

with a width and height of 201 pixels. The default user space coordinate system will have upper left corner at (0.0,0.0) and bottom right corner at (200.0,200.0). The face fits in a box with top left corner at (10,10), width 180, and height 180. The eye boxes have upper left corners at (40,50) and (130,50) with width 30 and height 30. The nose will be a triangle with vertices at (100,80), (90,110), and (110,110). The mouth will be the lower half of the ellipse that fits in the box with top left corner at (50,120), width 100, and height 40.

5.7.2 No-frills happy face

First we use the box description to produce the following geometrical description of the face.

```
    Ellipse2D.Double face = new Ellipse2D.Double(10,10,180,180);
```

```
Ellipse2D.Double leftEye = new Ellipse2D.Double(40,50,30,30);
Ellipse2D.Double rightEye = new Ellipse2D.Double(130,50,30,30);
Arc2D.Double mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);
Point2D.Double noseTop = new Point2D.Double(100,80);
Point2D.Double noseLeft = new Point2D.Double(90,110);
Point2D.Double noseRight = new Point2D.Double(110,110);
Line2D.Double nose1 = new Line2D.Double(noseTop,noseLeft);
Line2D.Double nose2 = new Line2D.Double(noseLeft,noseRight);
Line2D.Double nose3 = new Line2D.Double(noseRight,noseTop);
```

The coordinates directly correspond to those given in Figure 5.12. We have defined 5 geometrical objects (face, left eye, right eye, nose, mouth). The next step is to specify attributes for each of these objects and render them using `draw` and `fill` commands. For our first attempt we use the default black color and various line thicknesses. The statements are

```
g2D.setStroke(new BasicStroke(2.0f));
g2D.draw(face);
```

to obtain the face outlined in black with line thickness of 2 pixels,

```
g2D.fill(leftEye);
g2D.fill(rightEye);
```

to obtain the left and right eyes filled in black,

```
g2D.setStroke(new BasicStroke(4.0f));
g2D.draw(mouth);
```

to draw the mouth with a line thickness of 4 pixels, and

```
g2D.setStroke(new BasicStroke(2.0f));
g2D.draw(nose1);
g2D.draw(nose2);
g2D.draw(nose3);
```

to draw the nose as three lines forming a triangle with line thickness of 2 pixels. Here is the complete class `FaceMaker1` that produces the no-frills happy face.

Class `FaceMaker1`

book-projects/chapter5/happy_faces

```
package chapter5.happy_faces; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * First No-frills version of face maker to test the design.
 */
public class FaceMaker1 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      // Define the geometry (face, eyes, mouth, nose) in the default
      // user space coordinate system

      Ellipse2D.Double face = new Ellipse2D.Double(10,10,180,180);
      Ellipse2D.Double leftEye = new Ellipse2D.Double(40,50,30,30);
      Ellipse2D.Double rightEye = new Ellipse2D.Double(130,50,30,30);
      Arc2D.Double mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);
      Point2D.Double noseTop = new Point2D.Double(100,80);
      Point2D.Double noseLeft = new Point2D.Double(90,110);
      Point2D.Double noseRight = new Point2D.Double(110,110);
      Line2D.Double nose1 = new Line2D.Double(noseTop,noseLeft);
      Line2D.Double nose2 = new Line2D.Double(noseLeft,noseRight);
      Line2D.Double nose3 = new Line2D.Double(noseRight,noseTop);

      // Now render the geometry using draw, fill, various line thicknesses

      g2D.setStroke(new BasicStroke(2.0f));
      g2D.draw(face);

      g2D.fill(leftEye);
      g2D.fill(rightEye);

      g2D.setStroke(new BasicStroke(4.0f));
      g2D.draw(mouth);

      g2D.setStroke(new BasicStroke(2.0f));
      g2D.draw(nose1);
      g2D.draw(nose2);
      g2D.draw(nose3);
   }

   public void draw()
   {
      new GraphicsFrame("Happy Face 1", new FaceMaker1(), 201, 201);
   }

   public static void main(String[] args)
   {
      new FaceMaker1().draw();
   }
}
```

The output window is shown in Figure 5.13.

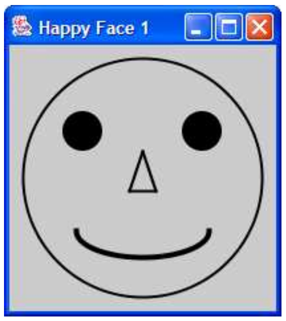


Figure 5.13: `FaceMaker1`: a no-frills happy face

5.7.3 Colorful happy face

We can improve on our happy face by using colors and fills for the parts of the face. The statements are

```
g2D.setPaint(Color.pink);
g2D.fill(face);
```

to fill the face with pink,

```
g2D.setPaint(Color.black);
g2D.setStroke(new BasicStroke(2.0f));
g2D.draw(face);
```

to outline it in black, 2 pixels wide,

```
g2D.setPaint(Color.blue);
g2D.fill(leftEye);
g2D.fill(rightEye);
```

to fill the eyes with blue, and

```
g2D.setPaint(Color.red);
g2D.setStroke(new BasicStroke(4.0f));
g2D.draw(mouth);
```

to give the face a red mouth 4 pixels wide.

Finally, to fill the nose with green we run into a problem. The `fill` method requires an argument that is an object in the `Shape` hierarchy shown in Figure 5.6. The triangular nose isn't defined that way. We need to make the nose into a `Shape` object. If you look at Figure 5.6 you will see that there is no triangle shape but there is a class called `GeneralPath` at the bottom. This class can be used to make custom shapes. To turn the nose into a triangular path, replace the statements that define the nose (the three `Point2D` statements and three `Line2D` statements) by the statements

```
GeneralPath nose = new GeneralPath();
nose.moveTo(100.0F,80.0F);  // start at top of nose
nose.lineTo(90.0F,110.0F);  // line to bottom left
nose.lineTo(110.0F,110.0F); // line to bottom right
nose.closePath();           // draws line back to top of nose
```

These statements first create an empty `GeneralPath` object called `nose`. This object has `moveTo` and `lineTo` methods which can be used to define the geometry of the custom shape. These methods require floating point arguments rather than double precision ones. We use the `moveTo` command to indicate that we want to start defining the path at the top of the nose. Then we use a `lineTo` to define a line from the top to the bottom left corner of the triangle, and another `lineTo` to define a horizontal line from there to the bottom right corner of the triangle. Finally the `closePath` method is used to close the path by defining a line back to the beginning. We now have a closed geometrical path that defines our nose so we can fill it, or draw it, or both. To fill it with green use

```
g2D.setPaint(Color.green);
g2D.fill(nose);
```

Here is the complete class for drawing the colorful happy face.

Class `FaceMaker2`

book-projects/chapter5/happy_faces

```
package chapter5.happy_faces; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Improved version of FaceMaker1 that uses colors and a shape for the nose.
 */
public class FaceMaker2 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      // Define the geometry (face, eyes, mouth, nose) in the default
      // user space coordinate system

      Ellipse2D.Double face = new Ellipse2D.Double(10,10,180,180);
      Ellipse2D.Double leftEye = new Ellipse2D.Double(40,50,30,30);
      Ellipse2D.Double rightEye = new Ellipse2D.Double(130,50,30,30);
      Arc2D.Double mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);

      // Make a custom shape for the nose so we can fill it
      // Note that paths use float type not double

      GeneralPath nose = new GeneralPath();
      nose.moveTo(100.0f,80.0f);  // start at top of nose
      nose.lineTo(90.0f,110.0f);  // line to bottom left
      nose.lineTo(110.0f,110.0f); // line to bottom right
```

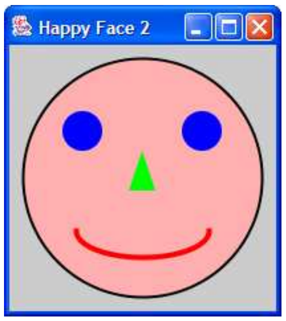


Figure 5.14: `FaceMaker2`: a colorful happy face

```
      nose.closePath();           // draws line back to top of nose

      // Now render the geometry using draw, fill, various
      // line thicknesses, and color attributes

      g2D.setPaint(Color.pink);
      g2D.fill(face);
      g2D.setPaint(Color.black);
      g2D.setStroke(new BasicStroke(2.0f));
      g2D.draw(face);

      g2D.setPaint(Color.blue);
      g2D.fill(leftEye);
      g2D.fill(rightEye);

      g2D.setPaint(Color.red);
      g2D.setStroke(new BasicStroke(4.0f));
      g2D.draw(mouth);

      g2D.setStroke(new BasicStroke(2.0f));
      g2D.setPaint(Color.green);
      g2D.fill(nose);
   }

   public void draw()
   {
      new GraphicsFrame("Happy Face 2", new FaceMaker2(), 201, 201);
   }

   public static void main(String[] args)
   {
      new FaceMaker2().draw();
   }
}
```

The output window is shown in Figure 5.14.

5.7.4 Facial transformations

After defining a geometric object, such as the happy face, it is possible to apply transformations to it. For example we can translate it to a new position on the drawing surface, scale it about the origin (0,0) so that it is larger or smaller, or even rotate it about (0,0). This would be difficult to do if we had to calculate all the new coordinates of the face. Imagine trying to draw the happy face in Figure 5.14 rotated by 45 degrees about its center by calculating the new coordinates of all the objects that make up the face. Fortunately it is not necessary to do this.

Affine transformations

Transformations such as translations, rotations, scalings, and their compositions are called affine transformations. They can be used to change from one coordinate system to another. We can also think of the transformation process as a transformation of the geometric objects themselves.

The graphics context, `Graphics2D`, that we have been using has an affine transformation associated with it. Initially this transformation is just the default one that transforms user space to device space (see Figure 5.2) in such a way that the drawing surface in the integer device space with origin $(0,0)$ at the top left and lower right corner at $(w-1,h-1)$ corresponds to a double precision user space with coordinates $(0.0,0.0)$ at the top left and $(w-1,h-1)$ at the bottom right. We say that user space is mapped to device space using the identity transformation.

So far in our program classes we have accepted this default transformation. However, we can modify it using affine transformations. Then all rendering commands will use the new transformation. It is important to realize that each time the `paintComponent` method is called by the Java run-time system this default transformation is provided in the graphics context.

Affine transformations are objects from the `AffineTransform` class. To construct a default affine transformation object called `at`, we use the statement

```
AffineTransform at = new AffineTransform();
```

Initially, this is the identity transformation.

Resizing the face to fit the window

Our happy face was designed in Figure 5.12 to fill a 201 by 201 pixel rectangle. If we resize the window the face does not change its size. It would be nice if we could scale the face so that it just fits the current window.

We can do this by modifying `FaceMaker2` to use an affine transformation to apply the appropriate scale factor. We will think in terms of transforming the face itself rather than the coordinate system.

As an example we start with Figure 5.15(a) which shows the 201 by 201 pixel face in a 301 by 301 pixel window. We can get the maximum coordinates of the window using

```
double xMax = getWidth() - 1;
double yMax = getHeight() - 1;
```

Now there are three steps to transforming the face.

1. Translate the face as shown in Figure 5.15(a) so that its center is at the top left corner $(0.0, 0.0)$. This means to subtract 100 from the x and y coordinates of each point, so that the original face center at $(100.0, 100.0)$ is now at $(0.0, 0.0)$. The result of this transformation is shown in Figure 5.15(b). Of course, only one quarter of the face is now visible.

2. Scale the face by the factor $xMax/200$ in the horizontal direction and the factor $yMax/200$ in the vertical direction (this gives a factor of 1.5 in each direction for our 301 by 301 pixel example). Since scaling takes place about the origin $(0, 0)$ this is why we first translated the center of the face to the origin.

   The result of the previous translation and this scaling about the origin is shown in Figure 5.15(c). The face is now larger but again only one quarter of the face is visible.

3. Finally, translate the face back so that its center is at $(xMax/2, yMax/2)$, the center of the window. Now the entire face is scaled to fit in the center of the window as shown in Figure 5.15(d).

To implement the composition of these three transformations in Java we construct an affine transformation object as follows.

```
AffineTransform at = new AffineTransform(); // identity transformation
at.translate(xMax / 2, yMax / 2);           // applied third
at.scale(xMax / 200, yMax / 200);           // applied second
at.translate(-100, -100);                   // applied first
```

Here we use the `translate` and `scale` methods to modify the affine transformation object `at`. The `scale` method always scales about the origin (0,0). That's why we needed to first translate the face so that its center was at (0,0).

It is important to understand that the transforms appear in the source code in the reverse order to the three steps shown above. You always do this when you are thinking of transforming objects instead of coordinate systems. It is called the "first applied last written" rule for the composition of object transformations. It is also important to note that the line thickness is also scaled, by 1.5 in the example.

The final step is to change the default transform in the graphics context `g2D` so that it uses our affine transform. This modification is done by the `transform` method in the statement

```
g2D.transform(at);
```

Now every time you draw a line, rectangle, ellipse or other object, the graphics context will apply this transform to your coordinates. Here is the class called `FaceMaker3` that uses this affine transformation:

Class `FaceMaker3`

book-projects/chapter5/happy_faces

```
package chapter5.happy_faces; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
```

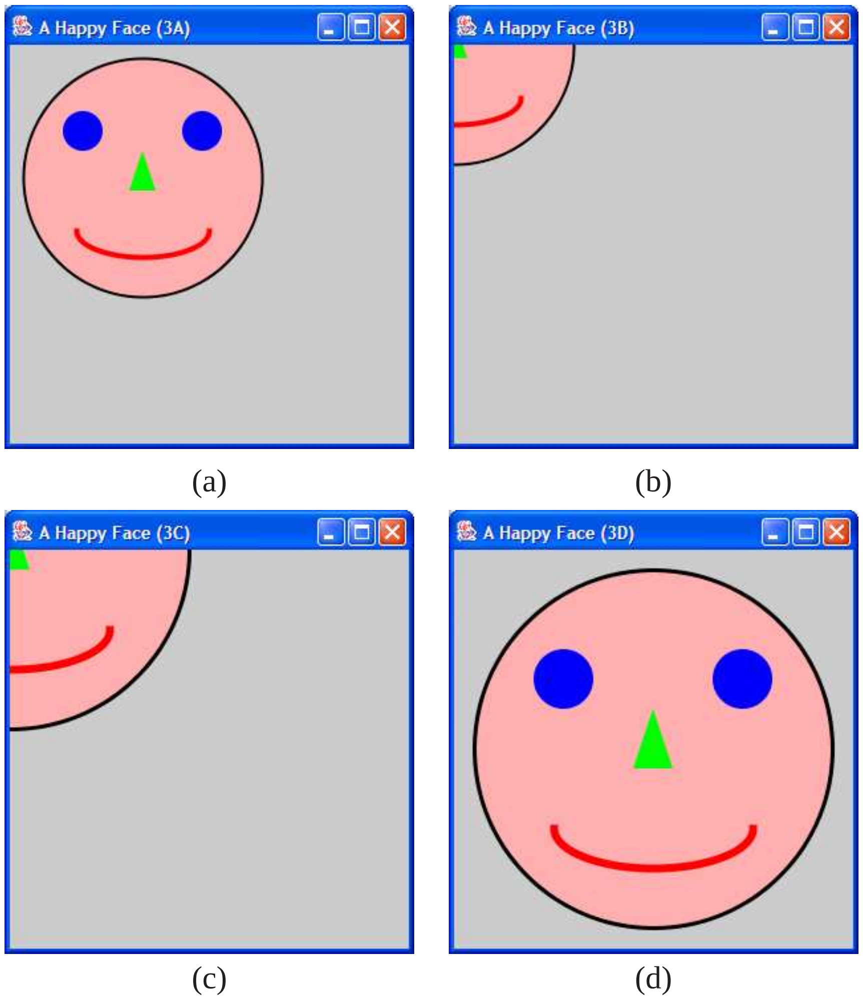


Figure 5.15: `FaceMaker3`: translating and scaling the happy face

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A variation of FaceMaker2 that uses affine transformations
 * to draw a face that scales with the size of the window
 */
public class FaceMaker3 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);
```

```
    // Define the geometry (face, eyes, mouth, nose) in the default
    // user space coordinate system

    Ellipse2D.Double face = new Ellipse2D.Double(10,10,180,180);
    Ellipse2D.Double leftEye = new Ellipse2D.Double(40,50,30,30);
    Ellipse2D.Double rightEye = new Ellipse2D.Double(130,50,30,30);
    Arc2D.Double mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);

    // Make a custom shape for the nose so we can fill it
    // Note that paths use float type not double

    GeneralPath nose = new GeneralPath();
    nose.moveTo(100.0f,80.0f);  // start at top of nose
    nose.lineTo(90.0f,110.0f);  // line to bottom left
    nose.lineTo(110.0f,110.0f); // line to bottom right
    nose.closePath();           // draws line back to top of nose

    /*
     * Transform the geometry so that the face always has the size of
     * the window. The transformations are applied in the reverse order
     * of appearance.
     */
    double xMax = getWidth() - 1; // width of window in user space
    double yMax = getHeight() - 1; // height of window in user space
    AffineTransform at = new AffineTransform();
    at.translate(xMax / 2, yMax / 2); // (4) translate back to center
    at.scale(xMax / 200, yMax / 200); // (2) scale to fit window size
    at.translate(-100, -100);         // (1) move face center to (0,0)
    g2D.transform(at);

    // Now render the geometry using draw, fill, various
    // line thicknesses, and color attributes

    g2D.setPaint(Color.pink);
    g2D.fill(face);
    g2D.setPaint(Color.black);
    g2D.setStroke(new BasicStroke(2.0f));
    g2D.draw(face);

    g2D.setPaint(Color.blue);
    g2D.fill(leftEye);
    g2D.fill(rightEye);

    g2D.setPaint(Color.red);
    g2D.setStroke(new BasicStroke(4.0f));
    g2D.draw(mouth);

    g2D.setStroke(new BasicStroke(2.0f));
    g2D.setPaint(Color.green);
    g2D.fill(nose);
}
```

```
   public void draw()
   {
      new GraphicsFrame("A Happy Face (3)", new FaceMaker3(), 301, 301);
   }

   public static void main(String[] args)
   {
      new FaceMaker3().draw();
   }
}
```

This class is identical to program `FaceMaker2` except for the statements that calculate the affine transformation. None of the coordinates of the geometry defining the original face needed to be changed.

Making a half-size face

Suppose we want a face that scales with the window size but is only half the maximum size. This can be done by modifying `FaceMaker3` to use the following affine transformation.

```
AffineTransform at = new AffineTransform();
at.translate(xMax / 2, yMax / 2); // (4) translate back to center
at.scale(xMax / 200, yMax / 200); // (3) scale to fit window size
at.scale(0.5, 0.5);               // (2) scale by 1/2
at.translate(-100, -100);         // (1) move face center to (0,0)
g2D.transform(at);
```

which can be simplified by combining the two consecutive scaling transformations to obtain the scaling transformation

```
at.scale(xMax / 400, yMax / 400);
```

If you call the resulting class `FaceMaker4` the result is shown in Figure 5.16.

Rotated happy face

Suppose we want a full-size face that scales with the window size but is rotated 45 degrees clockwise This can be done by modifying `FaceMaker3` to use the following affine transformation.

```
AffineTransform at = new AffineTransform();
at.translate(xMax / 2, yMax / 2); // (4) translate back to center
at.scale(xMax / 200, yMax / 200); // (3) scale to fit window size
at.rotate(Math.toRadians(45));    // (2) rotate 45 degrees
at.translate(-100, -100);         // (1) move face center to (0,0)
g2D.transform(at);
```

If you call the resulting class `FaceMaker5` the result is shown in Figure 5.17. The normal convention for positive rotation angles is counterclockwise but it is clockwise here because the device coordinate system has *y* coordinate that increased from top to bottom. To obtain a counterclockwise rotation use a negative angle.

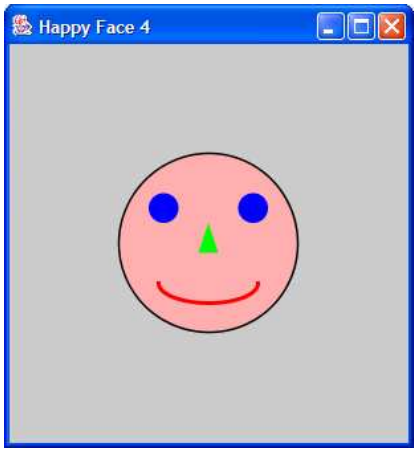


Figure 5.16: `FaceMaker4`: a half-size happy face

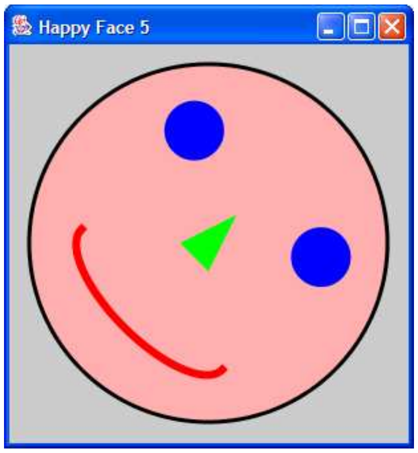


Figure 5.17: `FaceMaker5`: a rotated happy face

5.7.5 Four happy faces for the price of one

As a final example of object transformations let us modify `FaceMaker3` to draw four half size happy faces, in a two by two arrangement, on the 201 by 201 pixel device space with centers at $(xMax/4, yMax/4)$, $(3*xMax/4, yMax/4)$, $(xMax/4, 3*yMax/4)$, $(3*xMax/4, 3*yMax/4)$. (see Figure 5.18). Four affine transforms are needed here. The procedure is to define one transform, draw the face, redefine this transform, draw the face, and so on.

Top-level description

To begin, the class has the following structure

```
public class FaceMaker6 extends JPanel
{
```

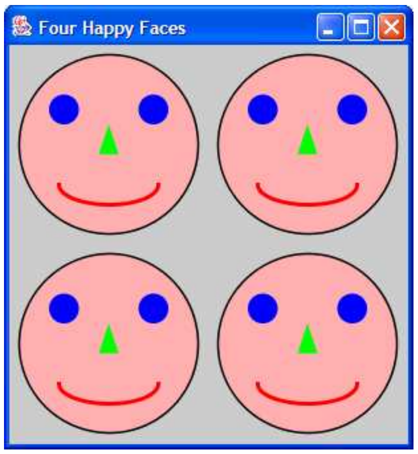


Figure 5.18: `FaceMaker6`: four happy faces

```
    private Ellipse2D.Double face, leftEye, rightEye;
    private Arc2D.Double mouth;
    private GeneralPath nose;

    public void paintComponent(Graphics g)
    {
       super.paintComponent(g);
       Graphics2D g2D = (Graphics2D) g;
       g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON);

       // Draw four faces in 2 by 2 arrangement using translations to
       // position the faces

       double xMax = getWidth() - 1;
       double yMax = getHeight() - 1;
       drawFace(g2D,xMax/4, yMax/4);   // with center at (xMax/4, yMax/4)
       drawFace(g2D,3*xMax/4, yMax/4);   // with center at (3*xMax/4, yMax/4)
       drawFace(g2D,xMax/4, 3*yMax/4);   // with center at (xMax/4, 3*yMax/4)
       drawFace(g2D,3*xMax/4, 3*yMax/4);   // with center at (3*xMax/4, yMax/4)
    }

    public static void main(String[] args)
    {
       new GraphicsFrame("Four Happy Faces", new FaceMaker6(), 301, 301);
    }
 }
```

where we now use data fields for the geometrical objects that make up the face: we will refer to them outside the `paintComponent` method so they can no longer be local to that method as in previous versions of the class.

Drawing one face

This structure clearly indicates that we need a `drawFace` method that takes the graphics context and the *x* and *y* coordinates of the face center as actual arguments. This method is responsible for defining the face geometry, setting up the proper affine transformation using the coordinates of the face center and rendering the face, so it has the structure.

```
private void drawFace(Graphics2D g2D, double x, double y)
{
   // (1) define the face geometry
   // (2) save the default transformation in g2D
   // (3) set up an affine transform for face centered at (x,y)
   // (4) use it to modify the default transformation
   // (5) render the face using the this transformation
   // (6) restore the default transformation in g2D
}
```

We now consider each of these six steps in turn. We first define the geometry as before using the default coordinate system. Now we write a method to do this having the prototype

```
private void defineFaceGeometry()
```

Since the `drawFace` method will be called several times, and each time it is called it is assumed that the default transformation is in effect, it is very important that we first save the default transformation, before making local changes, and then restore it after rendering the face. The saving and restoring can be done with the `getTransform` and `setTransform` methods in the graphics context, so steps (2) and (6) are easy.

To make our modifications to the default transformation we will use a method having the prototype

```
private AffineTransform getFaceTransform(double x, double y)
```

This method constructs and returns the appropriate affine transform using the specified coordinates for the center of the face.

To render the face using various colors and line thicknesses we now use a method with prototype

```
private void renderFace(Graphics2D g2D)
```

since it needs to know the graphics context.

Therefore the next level in the design process is complete and the `drawFace` method is given by

```
private void drawFace(Graphics2D g2D, double x, double y)
{
   defineFaceGeometry();
```

```
      AffineTransform save = g2D.getTransform();
      AffineTransform at = getFaceTransform(x, y);
      g2D.transform(at);
      renderFace(g2D);
      g2D.setTransform(save);  // restore default transform
   }
```

The two statements that construct the local transformation `at` and use it to modify the default one could also have been expressed as the single statement

```
   g2D.transform(getFaceTransform(x, y));
```

Defining the face geometry

The `defineFaceGeometry` method is given by

```
   private void defineFaceGeometry()
   {
      face = new Ellipse2D.Double(10,10,180,180);
      leftEye = new Ellipse2D.Double(40,50,30,30);
      rightEye = new Ellipse2D.Double(130,50,30,30);
      mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);
      nose = new GeneralPath();
      nose.moveTo(100.0f,80.0f);  // start at top of nose
      nose.lineTo(90.0f,110.0f);  // line to bottom left
      nose.lineTo(110.0f,110.0f); // line to bottom right
      nose.closePath();           // draws line back to top of nose
   }
```

The only difference here is that we must not re-declare the object variables. They have already been declared as data fields.

Transforming a face

The `getFaceTransform` method uses the ideas from `FaceMaker3` and is given by

```
   private AffineTransform getFaceTransform(double x, double y)
   {
      double xMax = getWidth() - 1;
      double yMax = getHeight() - 1;
      AffineTransform at = new AffineTransform();
      at.translate(x, y);               // (4) translate object back to (x,y)
      at.scale(0.5,0.5);                // (3) scale object by one half
      at.scale(xMax / 200, yMax / 200); // (2) scale to fit window
      at.translate(-100,-100);          // (1) translate object to origin
      return at;
   }
```

It constructs an affine transform and returns a reference to it. Here steps (1) and (2) are the same as in `FaceMaker3`, and the only difference is that the final translation uses `x` and `y` to position the half-size face.

Rendering a face

Finally, the geometry is rendered as before by the method

```
private void renderFace(Graphics2D g2D)
{
   g2D.setPaint(Color.pink);
   g2D.fill(face);
   g2D.setPaint(Color.black);
   g2D.setStroke(new BasicStroke(2.0f));
   g2D.draw(face);

   g2D.setPaint(Color.blue);
   g2D.fill(leftEye);
   g2D.fill(rightEye);

   g2D.setPaint(Color.red);
   g2D.setStroke(new BasicStroke(4.0f));
   g2D.draw(mouth);

   g2D.setPaint(Color.green);
   g2D.fill(nose);
}
```

We made the geometrical objects into data fields so that `defineFaceGeometry` and `renderFace` can refer to them. If they had been declared in `defineFaceGeometry` they would have been local to that method. Putting everything together gives the complete program class.

Class `FaceMaker6`

book-projects/chapter5/happy_faces

```
package chapter5.happy_faces; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Four for the price of one. Using affine transformations
 * to draw four copies of the  half-size face from FaceMaker3
 */
public class FaceMaker6 extends JPanel
{
   // Data fields are geometrical objects defining the face
```

```
private Ellipse2D.Double face, leftEye, rightEye;
private Arc2D.Double mouth;
private GeneralPath nose;

public void paintComponent(Graphics g)
{
   super.paintComponent(g);
   Graphics2D g2D = (Graphics2D) g;
   g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

   // Draw four faces in 2 by 2 arrangement using translations to
   // position the faces

   double xMax = getWidth() - 1;
   double yMax = getHeight() - 1;
   drawFace(g2D, xMax/4, yMax/4);   // with center at (xMax/4, yMax/4)
   drawFace(g2D, 3*xMax/4, yMax/4);  // with center at (3*xMax/4, yMax/4)
   drawFace(g2D, xMax/4, 3*yMax/4);  // with center at (xMax/4, 3*Ymax/4)
   drawFace(g2D, 3*xMax/4, 3*yMax/4); // with center at (3*xMax/4, 3*yMax/4)
}

// Draw the face with center at (x,y) using the specified
// graphics context

private void drawFace(Graphics2D g2D, double x, double y)
{
   // (1) define geometry, (2) save default transformation
   // since we will be calling this method several times
   // to make transformations relative to the default
   // transformation. (3) make a local transformation at,
   // (4) apply it to the default transformation, (5) render
   // the face geometry using this default transformation,
   // (6) restore the default transformation

   defineFaceGeometry();
   AffineTransform save = g2D.getTransform();
   AffineTransform at = getFaceTransform(x, y);
   g2D.transform(at);
   renderFace(g2D);
   g2D.setTransform(save); // restore default transform
}

private void defineFaceGeometry()
{
   // Define the geometry (face, eyes, mouth, nose) in the default
   // user space coordinate system

   face = new Ellipse2D.Double(10,10,180,180);
   leftEye = new Ellipse2D.Double(40,50,30,30);
   rightEye = new Ellipse2D.Double(130,50,30,30);
```

```
      mouth = new Arc2D.Double(50,120,100,40,180,180,Arc2D.OPEN);

      // Make a custom shape for the nose so we can fill it
      // Note that paths use float type not double

      nose = new GeneralPath();
      nose.moveTo(100.0f,80.0f);  // start at top of nose
      nose.lineTo(90.0f,110.0f);  // line to bottom left
      nose.lineTo(110.0f,110.0f); // line to bottom right
      nose.closePath();           // draws line back to top of nose
   }

   // Render the face using the current transformation

   private void renderFace(Graphics2D g2D)
   {
      g2D.setPaint(Color.pink);
      g2D.fill(face);
      g2D.setPaint(Color.black);
      g2D.setStroke(new BasicStroke(2.0f));
      g2D.draw(face);

      g2D.setPaint(Color.blue);
      g2D.fill(leftEye);
      g2D.fill(rightEye);

      g2D.setPaint(Color.red);
      g2D.setStroke(new BasicStroke(4.0f));
      g2D.draw(mouth);

      g2D.setPaint(Color.green);
      g2D.fill(nose);
   }

   // Construct an affine transformation that will scale the
   // face about its center and translate it so that the
   // center is at (x,y)

   private AffineTransform getFaceTransform(double x, double y)
   {
      double xMax = getWidth() - 1;
      double yMax = getHeight() - 1;
      AffineTransform at = new AffineTransform();
      at.translate(x, y);              // (4) translate object back to (x,y)
      at.scale(0.5,0.5);               // (3) scale object by one half
      at.scale(xMax / 200, yMax / 200); // (2) scale to fit window
      at.translate(-100,-100);         // (1) translate object to origin
      return at;
   }

   public void draw()
   {
      new GraphicsFrame("Four Happy Faces", new FaceMaker6(), 301, 301);
   }


   public static void main(String[] args)
   {
      new FaceMaker6().draw();
   }
}
```

5.7.6 Running the six face maker programs together

Since the `GraphicsFrame` class and the six face maker classes all construct objects it is possible to run them all together using the following runner class.

Class `AllTogether`

book-projects/chapter5/happy_faces

```
package chapter5.happy_faces; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages

/**
 * Display six happy face frames all together
 */
public class AllTogether
{
   public void draw()
   {
      new GraphicsFrame("FaceMaker6", new FaceMaker6(), 701, 701);
      new GraphicsFrame("FaceMaker5", new FaceMaker5(), 601, 601);
      new GraphicsFrame("FaceMaker4", new FaceMaker4(), 501, 501);
      new GraphicsFrame("FaceMaker3", new FaceMaker3(), 401, 401);
      new GraphicsFrame("FaceMaker2", new FaceMaker2(), 301, 301);
      new GraphicsFrame("FaceMaker1", new FaceMaker1(), 201, 201);
   }

   public static void main(String[] args)
   {
      new AllTogether().draw();
   }
}
```

5.8 Making your own coordinate transformations

The coordinate system for the default user space, corresponding to the identity transformation, is called the **default coordinate system**. It is rarely a useful system for real applications. For example, we might be developing a bar graph program with 4 vertical bars, one for each quarter of the year, and height given by the net profit in dollars for a company for each quarter. The natural coordinate system here has a horizontal coordinate ranging from 0 to 5, allowing the bars to be centered around 1 to 4 with a little space on either side of each bar if desired, and a vertical coordinate ranging from 0 to 10000, increasing from bottom to top. This is an example of a **world coordinate system** or a problem domain coordinate system. For the default system we would require a drawing surface that is 5 pixels wide and 10000 pixels (10 feet) tall!

As another example we may want to draw the graph of one period of the sine curve $y = \sin x$. Here the x-axis would range from 0 to 2π and the y-axis would range from -1 to 1.

Thus, we need to be able to convert coordinates from the world coordinate system to the default user coordinate system. There are several ways to do this:

1. Develop the specific transformations for each program and apply them directly.
2. Develop the general transformation from the world to the default user system, which can be customized for use in any program.
3. Use the affine transformations that are part of Java 2D to obtain the general transformation from the world to the default user system.

We will discuss each method.

5.8.1 Specific transformations

Continuing the bar graph example, we need to convert the bar graph world coordinate system to the default user coordinate system, as shown in Figure 5.19. Here the world coordinate system

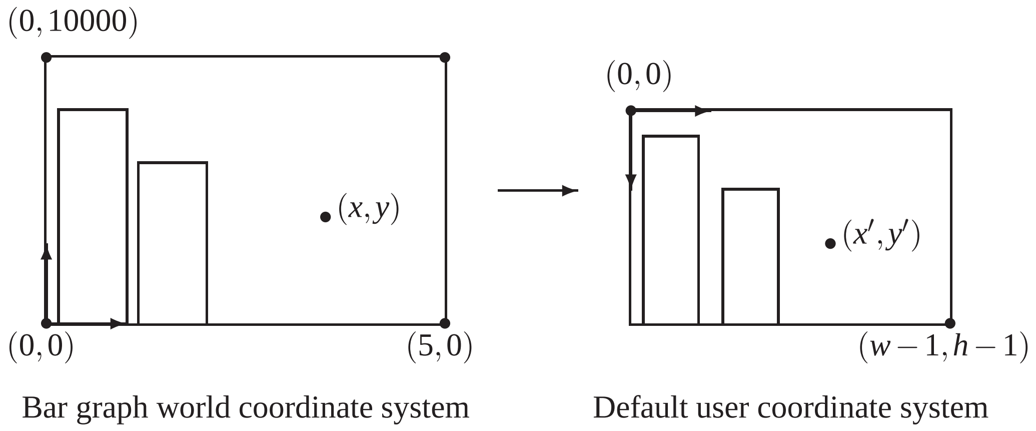

$(0,10000)$
(x,y)
$(0,0)$
$(5,0)$
Bar graph world coordinate system
$(0,0)$
(x',y')
$(w-1,h-1)$
Default user coordinate system

Figure 5.19: Bar graph coordinate transformation

is a right handed coordinate system with origin at the bottom left corner. A typical point has coordinates (x,y). The default user system is a left handed coordinate system with origin at the top left corner. A typical point has coordinates (x',y'). To transform the x coordinate we use the fact that the range 0 to 5 must be transformed to the interval 0 to $w-1$, where w is the width in pixels of the screen window. Similarly, the y coordinate in the range 0 to 10000 (bottom to top) must be transformed to the range $h-1$ to 0 (bottom to top), where h is the height in pixels of the screen window. The transformation equations are

$$\begin{aligned} x' &= s_x x, \text{ where } s_x = \left(\frac{w-1}{5}\right) \\ y' &= h-1-s_y y, \text{ where } s_y = \left(\frac{h-1}{10000}\right) \end{aligned}$$

As a check, these equations clearly show that $x=0$ is transformed to $x'=0$, $x=5$ is transformed to $x'=w-1$, $y=0$ is transformed to $y'=h-1$, and $y=10000$ is transformed to $y'=0$.

Let us write a very simple program that draws four vertical bars using this coordinate transformation. We suppose that the bar values are 8000.0, 10000.0, 5000.0, and 2000.0. First we write a local method called `bar` that returns a `Rectangle2D.Double` object representing one of the bars. It needs the prototype

```
private Rectangle2D.Double bar(int barNumber, double barHeight,
   int w, int h)
```

where `barNumber` takes on the values 0, 1, 2, and 3, `barHeight` is the height of the bar in the world coordinate system, and `w` and `h` are the width and height of the drawing surface in pixels. We can call it four times to define the bars using the statements

```
int w = getWidth(); // width of window in pixels;
int h = getHeight(); // height of window in pixels;

Rectangle2D.Double r1 = bar(0, 8000.0, w, h);
Rectangle2D.Double r2 = bar(1, 10000.0, w, h);
Rectangle2D.Double r3 = bar(2, 5000.0, w, h);
Rectangle2D.Double r4 = bar(3, 2000.0, w, h);
```

Now we can fill the bars with colors,

```
g2D.setPaint(Color.red); g2D.fill(r1);
g2D.setPaint(Color.green); g2D.fill(r2);
g2D.setPaint(Color.blue); g2D.fill(r3);
g2D.setPaint(Color.white); g2D.fill(r4);
```

and outline them in black.

```
g2D.setPaint(Color.black);
g2D.draw(r2);
g2D.draw(r1);
g2D.draw(r3);
g2D.draw(r4);
```

The complete definition of the `bar` method is

```
     private Rectangle2D.Double bar(int barNumber, double barHeight, int w, int h)
     {
        double sx = (w-1) / 5.0;      // x scale factor
        double sy = (h-1) / 10000.0;   // y scale factor
        double xTopLeft = (0.5 + barNumber*1.0) * sx;
        double yTopLeft = (h-1) - sy * barHeight; // reverse y axis
        double barW = sx * 1.0;
        double barH = sy * barHeight;
        return new Rectangle2D.Double(xTopLeft, yTopLeft, barW, barH);
     }
```

Here the mathematical formulas are used to define the scale factors `sx` and `sy`. The width of each bar is chosen to be one unit in the world system. We have written `barNumber*1.0` to emphasize this, although `barNumber` could be used. Therefore, the bars will have no space between them, and they will have width `sx` in default user space. To convert a width and height to default user space, multiply it by `sx` and `sy`, respectively.

The top left *x* coordinates of the bars have the values 0.5, 1.5, 2.5, and 3.5, which can be expressed as `0.5 + barNumber*1.0`, so we can express them in default user space as `(0.5 + barNumber*1.0)*sx`. Notice that the *y* coordinates of the tops of the bars in default user space do not have the values `sy * barHeight`: since the *y* direction is reversed we have to subtract these values from `h-1`. Finally, the `bar` method returns a new `Rectangle2D.Double` object defined using the transformed default user space values. Here is the complete program class

Class `BarGraph1`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Illustrate the use of specific coordinate transformations
 * from the world coordinate system for a bar graph to default user
 * coordinates. This is not very easy to do for each special case.
 */
public class BarGraph1 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      int w = getWidth(); // width of window in pixels;
      int h = getHeight(); // height of window in pixels;

      Rectangle2D.Double r1 = bar(0, 8000.0, w, h);
      Rectangle2D.Double r2 = bar(1, 10000.0, w, h);
      Rectangle2D.Double r3 = bar(2, 5000.0, w, h);
      Rectangle2D.Double r4 = bar(3, 2000.0, w, h);

      g2D.setPaint(Color.red); g2D.fill(r1);
      g2D.setPaint(Color.green); g2D.fill(r2);
      g2D.setPaint(Color.blue); g2D.fill(r3);
      g2D.setPaint(Color.white); g2D.fill(r4);

      g2D.setPaint(Color.black);
      g2D.draw(r2);
      g2D.draw(r1);
      g2D.draw(r3);
      g2D.draw(r4);
   }

   /*
    * Default coord system: origin at top left corner, lower right corner at
    * (w-1, h-1) where w, h are width and height in pixels.
    * Bar coord system: origin in lower left corner, width 5 and height 10000.
    * Note: shape objects always assume origin is at top left corner of window.
    */
   private Rectangle2D.Double bar(int barNumber, double barHeight, int w, int h)
   {
      double sx = (w-1) / 5.0;      // x scale factor
      double sy = (h-1) / 10000.0;  // y scale factor
      double xTopLeft = (0.5 + barNumber*1.0) * sx;
      double yTopLeft = (h-1) - sy * barHeight; // reverse y axis
      double barW = sx * 1.0; // bar width
      double barH = sy * barHeight; // bar height
      return new Rectangle2D.Double(xTopLeft, yTopLeft, barW, barH);
   }

   public void draw()
   {
      new GraphicsFrame("Bar Graph By Hand", new BarGraph1(), 301, 201);
   }

   public static void main(String[] args)
   {
      new BarGraph1().draw();
   }
}
```

The output is shown in Figure 5.20.

5.8.2 World to default user transformation

It may be inconvenient to work out the details of the world to default user coordinate transformations for each program. A better approach is to derive the general transformations. Consider a world coordinate system with bottom left corner at (x_L, y_B) and upper right corner at (x_R, y_T), as

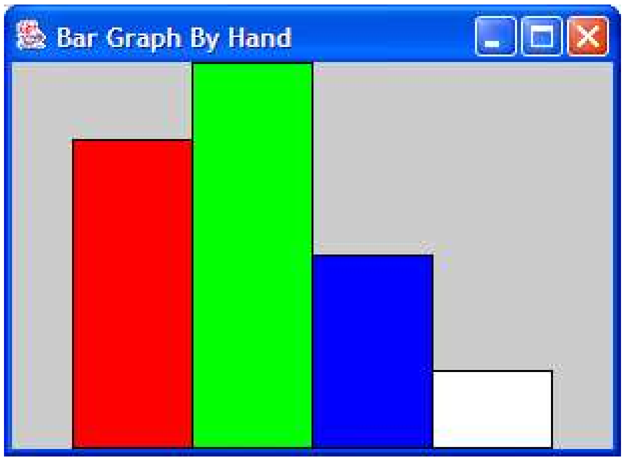


Figure 5.20: Output of the `BarGraph1` program

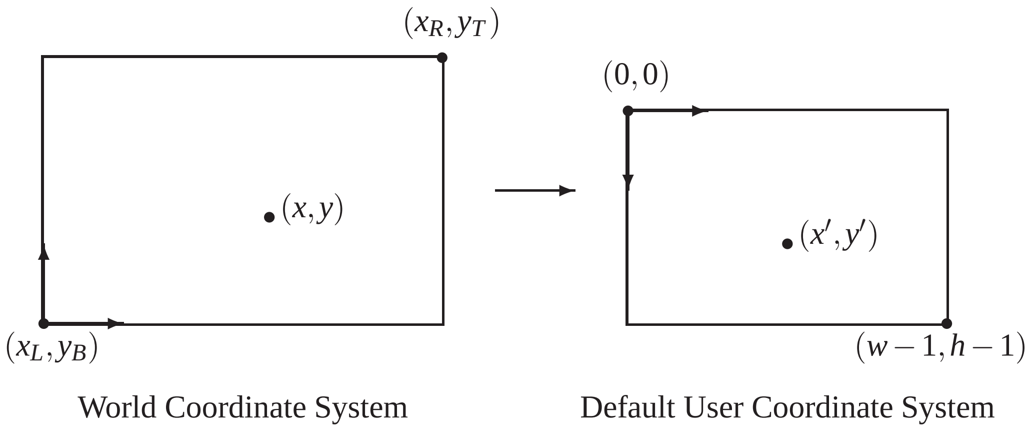


Figure 5.21: World to default user coordinate transformation

shown in Figure 5.21. We want to transform this world coordinate system to the default user coordinate system with origin at the top left. If w and h are the width and height of the screen window in pixels, then the lower right corner is at $(w-1, h-1)$. The transformation of the x coordinate of a point is obtained from the fact that the ratio of the distance of x from the left side of the rectangle to the width of the rectangle must be the same in both coordinate systems so

$$\frac{x \text{ distance}}{\text{rectangle width}} = \frac{x - x_L}{x_R - x_L} = \frac{x' - 0}{w - 1}$$

Similarly, for the y coordinate, the ratio of the distance of y from the bottom of the rectangle, to the height of the rectangle, must be the same in both coordinate systems so

$$\frac{y \text{ distance}}{\text{rectangle height}} = \frac{y - y_B}{y_T - y_B} = \frac{h - 1 - y'}{h - 1}$$

Again note that it is $h-1-y'$ not y' in the numerator because the y' coordinate origin is at the top and the distances in the ratios are positive.

We can solve these equations for x' and y' to obtain the general transformation equations:

$$\begin{aligned} x' &= s_x(x - x_L), \quad \text{where } s_x = \frac{w-1}{x_R - x_L} \\ y' &= s_y(y_T - y), \quad \text{where } s_y = \frac{h-1}{y_T - y_B} \end{aligned}$$

The factor $y_T - y$, rather than $y - y_T$, is due to the reversal of the direction of the y axis in going from world coordinates to device coordinates. The factors s_x and s_y are called the scale factors of the transformation.

5.8.3 Coordinate system class

An interesting object-oriented approach to implementing this world coordinate transformation is to consider a coordinate system transformation as an object that knows how to transform world coordinates to the default user coordinate system.

Each coordinate system transformation is defined by six numbers, namely the coordinates of the bottom left corner and top right corner of the world coordinate rectangle, and the width and height in pixels of the default user space. The constructor will need to specify these values and we need two methods to transform the x and y coordinates in the world system to the default user system. Therefore the class has the specification

```
public class CoordinateSystem
{
   // Construct a world coordinate system
   public CoordinateSystem(double xl, double xr, double yb, double yt,
      double w, double h) {...}
   // transform x coordinate from world to default user coordinates
   public double x(double x) {...}
   // transform y coordinate from world to default user coordinates
   public double y(double y) {...}
}
```

The complete class declaration is

Class `CoordinateSystem`

book-projects/custom_classes

```
package custom_classes; // remove this line if you’re not using packages
/**
 * A class to set up a window in a world coordinate system.
 * The window is defined by its lower left corner (xl,yb) and its upper
 * right corner (xr,yt), and it is mapped to a default user coordinate system
 * with origin (0,0) at the top left corner and lower right corner at (w-1,y-1).
 */
   public class CoordinateSystem
{
   private double xLeft, xRight;
   private double yBottom, yTop;
   private double width, height;
   private double scaleX, scaleY;

   /**
    * Construct a coordinate system and mapping from a world coordinate system
    * window defined by lower left corner (x1,yb) and upper right corner (xr,yt)
    * to a default user space window defined by origin (0,0) at top left corner
    * and lower right corner at (w-1,h-1).
    */
   public CoordinateSystem(double xl, double xr, double yb, double yt,
      double w, double h)
   {
      xLeft = xl;
      xRight = xr;
      yBottom = yb;
      yTop = yt;
      width = w;
      height = h;
      scaleX = (width - 1.0) / (xRight - xLeft);
      scaleY = (height - 1.0) / (yTop - yBottom);
   }

   /**
    * Map world coordinate x value to user space value.
    * @param x the world x coordinate to map
    * @return the x coordinate in user space.
    */
   public double x(double x)
   {
      return (x - xLeft) * scaleX;
   }

   /**
    * Map world coordinate y value to user space value.
    * @param y the world y coordinate to map
    * @return the y coordinate in user space.
    */
   public double y(double y)
   {
      return (yTop - y) * scaleY;
   }
}
```

The constructor arguments are the x range from x_L to x_R and the y range from y_B to y_T. The constructor just needs to assign values to the six data fields. For example, the statement

```
CoordinateSystem pixel = new CoordinateSystem(0,5,0,10000,301,201);
```

sets up a coordinate transformation for the bar graph shown in Figure 5.19. Then statements such as

```
double xp = pixel.x(x);
double yp = pixel.y(y);
```

can be used to transform the coordinates `(x,y)` of a point in the bar graph coordinate system to its coordinates `(xp,yp)` in default user space corresponding to the 301 by 201 pixel window.

5.8.4 Drawing a bar graph

We can easily rewrite the `BarGraph1` class to use the `CoordinateSystem` class. The result is

Class `BarGraph2`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import custom_classes.CoordinateSystem; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A version of BarGraph that uses a custom CoordinateSystem class
 * to convert the world coordinate system of the bar graph to device
 * coordinates. Every coordinate must be explicitly transformed by
 * the programmer. An alternate approach that avoids this is to
 * use AffineTransforms
 */
public class BarGraph2 extends JPanel
{
   CoordinateSystem pixel;

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      int w = getWidth();
      int h = getHeight();

      /* Set up a mapping from world to device coordinate systems.
       * In world system 0.0 <= x <= 5 (increasing to right)
       * and 0.0 <= y <= 10000 (increasing upward).
       * Here w, h are the width and height in pixels of JPanel
       */
      pixel = new CoordinateSystem(0.0, 5.0, 0.0, 10000.0, w, h);
      g2D.setStroke(new BasicStroke(1.0f));

      /* Specify world coordinates for top left corner of each rectangle,
       * it’s width, and it’s height.
       */
      Rectangle2D.Double r1 = bar(0.5, 8000.0, 1.0, 8000.0);
      Rectangle2D.Double r2 = bar(1.5, 10000.0, 1.0, 10000.0);
      Rectangle2D.Double r3 = bar(2.5, 5000.0, 1.0, 5000.0);
      Rectangle2D.Double r4 = bar(3.5, 2000.0, 1.0, 2000.0);

      g2D.setPaint(Color.red); g2D.fill(r1);
      g2D.setPaint(Color.green); g2D.fill(r2);
      g2D.setPaint(Color.blue); g2D.fill(r3);
      g2D.setPaint(Color.white); g2D.fill(r4);

      g2D.setPaint(Color.black);
      g2D.draw(r1);
      g2D.draw(r2);
      g2D.draw(r3);
      g2D.draw(r4);
   }

   /*
    * Transform rectangle with specified top left corner, width, height
    * into device coordinates (y increasing downward). Note how differences
    * are used to transform widths and heights.
    */
   private Rectangle2D.Double bar(
      double xMin, double yMax, double w, double h)
   {
      double width = pixel.x(w) - pixel.x(0);
      double height = pixel.y(0) - pixel.y(h); // note this difference
      return new Rectangle2D.Double(pixel.x(xMin), pixel.y(yMax), width, height);
   }

   public void draw()
   {
      new GraphicsFrame("Bar Graph Using CoordinateSystem",
         new BarGraph2(), 301, 201);
   }

   public static void main(String[] args)
   {
      new BarGraph2().draw();
   }
}
```

Here we have to be careful writing the `bar` method since a `Rectangle2D.Double` object is specified in the device coordinate system with origin at the top left corner and *y* coordinate increasing downward, whereas our coordinate system has the origin at the the bottom left corner with the *y* coordinate increasing upward.

5.8.5 Drawing a regular pentagon

As another example, let us write a program class that draws a regular pentagon. If the center of the pentagon is at $(0,0)$ and the radius is r, then the coordinates (x_k, y_k), $k = 0,1,2,3,4$ of the 5 vertices are

$$x_k = r\cos ka, \quad y_k = r\sin ka$$

where the angle a is 60 degrees which is $72\pi/180$ radians (divide a circle of radius r into 5 equal sectors of 72 degrees each). To center the pentagon let us choose a world coordinate system in which x and y both range from -5 to 5, with the radius chosen as 4. We can use a `CoordinateSystem` object defined by

```
      CoordinateSystem pixel =
         new CoordinateSystem(-5,5,-5,5,getWidth(),getHeight());
```

to transform it so that the pentagon center is the center of the screen window.

For example the *x* coordinate of vertex 1 in the world system is `r*Math.cos(a)` and in the default user system it is `pixel.x(r*Math.cos(a))`. Similarly the *y* coordinate `r*Math.sin(a)` becomes `pixel.y(r*Math.sin(a))`. Here is the class.

Class `DrawPentagon1`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import custom_classes.CoordinateSystem; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Explicit transformation of coordinates using the CoordinateSystem class
 * to draw a pentagon.
 */
public class DrawPentagon1 extends JPanel
{
   public void paintComponent(Graphics g)
```

```
{
   super.paintComponent(g);
   Graphics2D g2D = (Graphics2D) g;
   g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

   int w = getWidth();
   int h = getHeight();

   /* Set up a mapping from world to device coordinate systems.
    * In world system -5 <= x <= 5 (increasing to right)
    * and -5 <= y <= 5 (increasing upward).
    * Here w, h are the width and height in pixels of JPanel.
    */

   CoordinateSystem pixel = new CoordinateSystem(-5, 5, -5, 5, w, h);

    /* The pentagon will have its center at the origin of this
     * coordinate system.
     */

   double a = Math.toRadians(72.0); // 72 degrees in radians
   double r = 4.0; // radius of pentagon’s circumscribed circle

   /* Transform coordinates of vertices to default user space (pixels)
    * using pixel.x and pixel.y
    */

   double x0,y0,x1,y1,x2,y2,x3,y3,x4,y4;

   x0 = pixel.x(r*Math.cos(0*a));   y0 = pixel.y(r*Math.sin(0*a));
   x1 = pixel.x(r*Math.cos(1*a));   y1 = pixel.y(r*Math.sin(1*a));
   x2 = pixel.x(r*Math.cos(2*a));   y2 = pixel.y(r*Math.sin(2*a));
   x3 = pixel.x(r*Math.cos(3*a));   y3 = pixel.y(r*Math.sin(3*a));
   x4 = pixel.x(r*Math.cos(4*a));   y4 = pixel.y(r*Math.sin(4*a));

   // Construct a path for the pentagon in default user space

   GeneralPath pentagon = new GeneralPath();
   pentagon.moveTo((float) x0,(float) y0); // east
   pentagon.lineTo((float) x1,(float) y1);
   pentagon.lineTo((float) x2,(float) y2);
   pentagon.lineTo((float) x3,(float) y3);
   pentagon.lineTo((float) x4,(float) y4);
   pentagon.closePath();

   // fill it with yellow, then outline it in black using 2 pixel brush

   g2D.setPaint(Color.yellow);
   g2D.fill(pentagon);
   g2D.setPaint(Color.black);
   g2D.setStroke(new BasicStroke(2.0f));
```

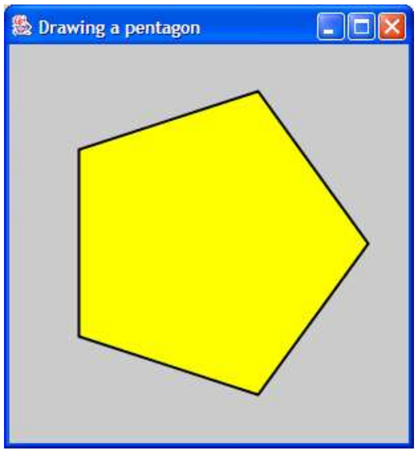


Figure 5.22: Output of the `DrawPentagon1` class

```
      g2D.draw(pentagon);
   }

   public void draw()
   {
      new GraphicsFrame("Drawing a pentagon",
         new DrawPentagon1(), 301, 301);
   }

   public static void main(String[] args)
   {
      new DrawPentagon1().draw();
   }
}
```

First the vertices are transformed using `cs.x` and `cs.y` to default user space and then the pentagon is constructed as a path. The `float` type must be used in the `moveTo` and `lineTo` method arguments.

Because the pentagon is a `Shape` object it can be used as an argument to the `draw` and `fill` methods of the graphics context. The output is shown in Figure 5.22.

5.8.6 General transformation using affine transformations

Another way to implement the general transformation is to construct an `AffineTransform` object as we did in some of the happy face programs. We can do this with the following `worldTransform` method that takes the *x* and *y* ranges `xMin` to `xMax` and `yMin` to `yMax` in the world coordinate system as arguments and the width and height of the drawing surface in pixels, namely `w` and `h`.

```
private AffineTransform worldTransform(double xMin, double xMax,
   double yMin, double yMax, int w, int h)
{
   double sx = (w-1) / (xMax - xMin); // scale factor in x direction
   double sy = (h-1) / (yMax - yMin); // scale factor in y direction
   AffineTransform at = new AffineTransform();
   at.scale(sx, -sy);     // -sy reverses y axis
   at.translate(-xMin, -yMax);  // upper left corner (xMin,yMax) to (0,0)
   return at;
}
```

Here we are thinking in terms of transforming a rectangular object from the world coordinate system to the default coordinate system as shown in Figure 5.23. The rectangle on the left in the

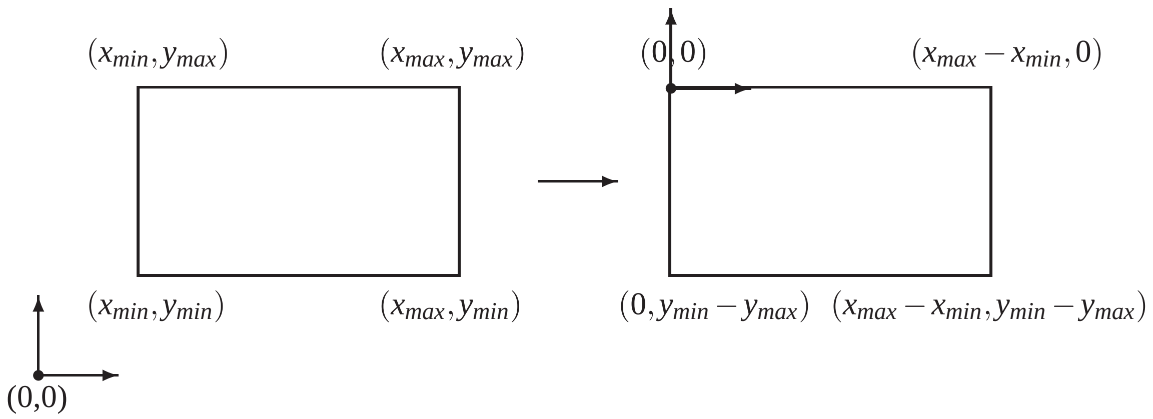


Figure 5.23: World to default user coordinate transformation

world coordinate system has lower left corner at (x_{min}, y_{min}) and upper right corner at (x_{max}, y_{max}).

We first need to translate this rectangle so that its top left corner is at the origin $(0,0)$. This is done by subtracting x_{min} from the x coordinates and y_{max} from the y coordinates. The affine transform which does this is

```
at.translate(-xMin, -yMax);
```

This gives the rectangle on the right side of Figure 5.23.

Now we scale this rectangle to the size of the rectangle shown on the right side of Figure 5.21. This is done using the scale factors

$$s_x = \frac{w-1}{x_{max} - x_{min}}, \qquad s_y = \frac{h-1}{y_{max} - y_{min}}$$

and we also need to change the direction of the y axis so that it points downward (this is a reflection in the x axis. The affine transform that does this scaling and reflection is

```
at.scale(sx, -sy);
```

Bar graph using an affine transformation

Here is a variation of the bar graph program that uses this method and the statements

```
    AffineTransform world = worldTransform(0.0, 5.0, 0.0, 10000.0, w, h);
    g2D.transform(world);
```

to define the world coordinate transform and apply it to the graphics context.

Class `BarGraph3`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * A version of BarGraph that uses a general affine
 * transformation to convert the world coordinate system of the
 * bar graph to device coordinates.
 */
public class BarGraph3 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      int w = getWidth();
      int h = getHeight();

      /* World coordinates: 0.0 <= x <= 5, 0.0 <= y <= 10000
       * w, h are width and height of window in pixels.
       */
      AffineTransform world = worldTransform(0.0, 5.0, 0.0, 10000.0, w, h);
      g2D.transform(world);

      // If you don’t want the line thickness to scale with respect to
      // the affine transformation simply specify a stroke width of 0.
      // Try something like 0.1f to see the effects of scaling

      g2D.setStroke(new BasicStroke(0.0f));

      /*
       * Since we now have a right handed coordinate system rectangles
       * are described using their lower left corner instead of the upper
       * left corner as would be the case in the left-handed device system.
       */

      Rectangle2D.Double r1 = new Rectangle2D.Double(0.5, 0.0, 1.0, 8000.0);
      Rectangle2D.Double r2 = new Rectangle2D.Double(1.5, 0.0, 1.0, 10000.0);
      Rectangle2D.Double r3 = new Rectangle2D.Double(2.5, 0.0, 1.0, 5000.0);
      Rectangle2D.Double r4 = new Rectangle2D.Double(3.5, 0.0, 1.0, 2000.0);
      g2D.setPaint(Color.red); g2D.fill(r1);
      g2D.setPaint(Color.green); g2D.fill(r2);
      g2D.setPaint(Color.blue); g2D.fill(r3);
      g2D.setPaint(Color.white); g2D.fill(r4);

      g2D.setPaint(Color.black);
      g2D.draw(r1);
      g2D.draw(r2);
      g2D.draw(r3);
      g2D.draw(r4);
   }

   /*
    * Transform right-handed coordinate system with lower left corner at
    * (xMin,yMin) and upper right corner at (xMax,yMax) to the left handed
    * device coordinate system with upper left corner at (0,0) and lower
    * right corner at (w-1,h-1) where w, h are the width and height of the
    * drawing window in pixels. In this coordinate system y increases
    * downwards.
    */
   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h)
   {
      double sx = (w-1) / (xMax - xMin); // scale factor in x direction
      double sy = (h-1) / (yMax - yMin); // scale factor in y direction
      AffineTransform at = new AffineTransform();
      at.scale(sx, -sy);     // -sy reverses y axis
      at.translate(-xMin, -yMax);  // upper left corner (xMin,yMax) to (0,0)
      return at;
   }

   public void draw()
   {
      new GraphicsFrame("Bar Graph Using AffineTransform",
         new BarGraph3(), 301, 201);
   }

   public static void main(String[] args)
   {
      new BarGraph3().draw();
   }
}
```

It is important to realize that when an affine transformation involves scaling the brush size is also scaled. Sometimes this is desirable, as in the face maker examples, but in the bar graph example it may not be desirable. If you don't want the line thickness to scale with respect to the affine transform simply specify a stroke width of 0:

```
g2D.setStroke(new BasicStroke(0.0f));
```

We have done this in the `BarGraph3` class. If this is not done the brush size will be huge because of the very different scales, 0 to 5 in the horizontal direction and 0 to 10000 in the vertical direction.

It is also important to note that we now define our shapes using world coordinates but, since this is a proper right-handed coordinate system, the frames of objects such as rectangles, ellipses, and arcs are now specified using their lower left corner rather than their upper left corner. For example, the first bar, `r1`, is specified using 0.5, 0.0 for the first two arguments since these are the coordinates of the lower left corner of the bar in the world coordinate system.

Using the affine transform method is usually better than the previous one using `pixel`, a `CoordinateSystem` object, since the later does not scale brush sizes and it requires the application of the `pixel.x` and `pixel.y` methods to every coordinate specified in your program. With the affine transformation approach we simply use our world coordinates everywhere and the graphics context will take care of the transformation to default user space.

Pentagon using an affine transformation

Here is a version of `DrawPentagon1` that uses an affine transformation:

Class `DrawPentagon2`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Using an affine transformation to set up a world coordinate system
 * and use it to transform the graphics context.
 */
public class DrawPentagon2 extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      int w = getWidth();
      int h = getHeight();

      /* Set up a mapping from world to device coordinate systems.
       * In world system -5 <= x <= 5 (increasing to right)
       * and -5 <= y <= 5 (increasing upward).
       * Here w, h are the width and height in pixels of JPanel.
       * Transform the graphics context to use this mapping.
       */

      AffineTransform world = worldTransform(-5, 5, -5, 5, w, h);
      g2D.transform(world);
      double a = Math.toRadians(72.0); // 72 degrees in radians
      double r = 4.0; // radius of circumscribed circle

      // Define the pentagon using world coordinates

      double x0,y0,x1,y1,x2,y2,x3,y3,x4,y4;

      x0 = r*Math.cos(0*a);   y0 = r*Math.sin(0*a);
      x1 = r*Math.cos(1*a);   y1 = r*Math.sin(1*a);
      x2 = r*Math.cos(2*a);   y2 = r*Math.sin(2*a);
      x3 = r*Math.cos(3*a);   y3 = r*Math.sin(3*a);
      x4 = r*Math.cos(4*a);   y4 = r*Math.sin(4*a);

      // Construct a path for the pentagon in default user space

      GeneralPath pentagon = new GeneralPath();
      pentagon.moveTo((float) x0,(float) y0); // east
      pentagon.lineTo((float) x1,(float) y1);
      pentagon.lineTo((float) x2,(float) y2);
      pentagon.lineTo((float) x3,(float) y3);
      pentagon.lineTo((float) x4,(float) y4);
      pentagon.closePath();

      // set the brush to two pixels.

      double pixelWidth  = Math.abs(1 / world.getScaleX()); // pixel width in world
      double pixelHeight = Math.abs(1 / world.getScaleY()); // pixel height in world

      // Now we can calculate a line thickness relative that is two pixels wide

      float thickness = 2 * (float) ( Math.min(pixelWidth, pixelHeight) );

      // Comment the following statement and you will see that the brush thickness
      // of the graphics context is transformed as well. It will be very thick.

      g2D.setStroke(new BasicStroke(thickness));

      // fill it with yellow, then outline it in black using 2 pixel brush

      g2D.setPaint(Color.yellow);
      g2D.fill(pentagon);
      g2D.setPaint(Color.black);
      g2D.draw(pentagon);
   }

   /*
    * Transform right-handed coordinate system with lower left corner at
    * (xMin,yMin) and upper right corner at (xMax,yMax) to the left handed
    * device coordinate system with upper left corner at (0,0) and lower
    * right corner at (w-1,h-1) where w, h are the width and height of the
    * drawing window in pixels. In this coordinate system y increases
    * downwards.
    */
   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h)
   {
      double sx = (w-1) / (xMax - xMin); // scale factor in x direction
      double sy = (h-1) / (yMax - yMin); // scale factor in y direction
      AffineTransform at = new AffineTransform();
      at.scale(sx, -sy);     // -sy reverses y axis
      at.translate(-xMin, -yMax);  // upper left corner (xMin,yMax) to (0,0)
      return at;
   }

   public void draw()
   {
      new GraphicsFrame("Drawing a pentagon (AffineTransform)",
         new DrawPentagon2(), 201, 201);
   }

   public static void main(String[] args)
   {
      new DrawPentagon2().draw();
   }
}
```

An interesting feature of this class is the calculation of the size of a pixel in the world coordinate system. Doing this lets us choose line thicknesses in terms of pixels so we can scale the brush independent of the affine transformation. In `DrawPentagon2` we have chosen a thickness of 2 pixels.

5.8.7 Transforming individual shapes

The `AffineTransform` class has a method called `createTransformedShape` with prototype

```
Shape createTransformedShape(Shape s)
```

That applies the affine transformation to the specified `Shape` object and returns the transformed `Shape` object. This provides another method for transforming objects without actually changing the default transformation associated with the graphics context.

For example, in the `DrawPentagon2` class we could define the world transformation

```
AffineTransform world = worldTransform(-5, 5, -5, 5, w, h);
```

and instead of using

```
g2D.transform(world);
```

we would define the pentagon as a `GeneralPath` object and use

```
pentagon = (GeneralPath) world.createTransformedShape(pentagon);
```

to transform the pentagon. Here is the revised class.

Class `DrawPentagon3`

book-projects/chapter5/coordinate_system

```
package chapter5.coordinate_system; // remove this line if you're not using packages
import custom_classes.GraphicsFrame; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Using an affine transformation to set up a world coordinate system.
 * This version does not transform the Graphics context. Instead it
 * uses createTransformedShape to transform world shapes to device shapes.
 * Since we are not transforming the graphics context the line thicknesses
 * will not change.
 */
public class DrawPentagon3 extends JPanel
{
   /**
    * Construct a pentagon drawing panel with a white background.
    */
   public DrawPentagon3()
   {
      setBackground(Color.white);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      int w = getWidth();
      int h = getHeight();

      /* Set up a mapping from world to device coordinate systems.
       * In world system -5 <= x <= 5 (increasing to right)
       * and -5 <= y <= 5 (increasing upward).
       * Here w, h are the width and height in pixels of JPanel.
       */

      AffineTransform world = worldTransform(-5, 5, -5, 5, w, h);

      double a = Math.toRadians(72.0); // 72 degrees in radians
      double r = 4.0; // radius of circumscribed circle

      // Define the pentagon using world coordinates.

      double x0,y0,x1,y1,x2,y2,x3,y3,x4,y4;

      x0 = r*Math.cos(0*a);   y0 = r*Math.sin(0*a);
      x1 = r*Math.cos(1*a);   y1 = r*Math.sin(1*a);
      x2 = r*Math.cos(2*a);   y2 = r*Math.sin(2*a);
      x3 = r*Math.cos(3*a);   y3 = r*Math.sin(3*a);
      x4 = r*Math.cos(4*a);   y4 = r*Math.sin(4*a);

      // Construct a path for the pentagon in default user space

      GeneralPath pentagon = new GeneralPath();
      pentagon.moveTo((float) x0,(float) y0); // east
      pentagon.lineTo((float) x1,(float) y1);
      pentagon.lineTo((float) x2,(float) y2);
      pentagon.lineTo((float) x3,(float) y3);
      pentagon.lineTo((float) x4,(float) y4);
      pentagon.closePath();

      // Now transform the pentagon shape from world to device coords

      pentagon = (GeneralPath) world.createTransformedShape(pentagon);

      // fill it with yellow, then outline it in black using 2 pixel brush

      g2D.setPaint(Color.yellow);
      g2D.fill(pentagon);
      g2D.setStroke(new BasicStroke(2.0f));
      g2D.setPaint(Color.black);
      g2D.draw(pentagon);
   }

   /*
    * Transform right-handed coordinate system with lower left corner at
    * (xMin,yMin) and upper right corner at (xMax,yMax) to the left handed
    * device coordinate system with upper left corner at (0,0) and lower
    * right corner at (w-1,h-1) where w, h are the width and height of the
    * drawing window in pixels. In this coordinate system y increases
    * downwards.
    */
   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h)
   {
      double sx = (w-1) / (xMax - xMin); // scale factor in x direction
      double sy = (h-1) / (yMax - yMin); // scale factor in y direction
      AffineTransform at = new AffineTransform();
      at.scale(sx, -sy);     // -sy reverses y axis
      at.translate(-xMin, -yMax);  // upper left corner (xMin,yMax) to (0,0)
      return at;
   }

   public void draw()
   {
      new GraphicsFrame("Drawing a pentagon (AffineTransform)",
         new DrawPentagon3(), 301, 301);
   }
   public static void main(String[] args)
   {
      new DrawPentagon3().draw();
   }
}
```

One advantage of this approach is that the line thickness will not be scaled since the affine transformation of the graphics context is not being changed. Another advantage is that several parts of a picture could be designed in different coordinate systems and transformed independently before drawing the complete picture.

5.9 Review exercises

► **Review Exercise 5.1** Define the following terms and give examples of each.

component	graphics context	device space
screen space	user space	default transformation
bounding rectangle	attribute	rendering
anti-aliasing	RGB color	affine transformation
translation	scaling	rotation
private method	default coordinate system	world coordinate system

► **Review Exercise 5.2** Explain the following methods or classes and write some Java statements to illustrate each.

`Shape`	`draw`	`fill`
`Point2D`	`getX`	`getY`
`Line2D`	`Rectangle2D`	`Ellipse2D`
`Arc2D`	`RoundRectangle2D`	`getBounds2D`
`paintComponent`	`JPanel`	`JComponent`
`GraphicsFrame`	`Graphics`	`Graphics2D`
`Color`	`setPaint`	`getPaint`
`BasicStroke`	`setStroke`	`getStroke`
`GeneralPath`	`moveTo`	`lineTo`
`closePath`	`AffineTransform`	`translate`
`scale`	`rotate`	`transform`
`getTransform`	`setTransform`	`CoordinateSystem`
`getWidth`	`getHeight`	

5.10 Programming exercises

► **Exercise 5.1 (A sad face maker)**
Write a class called `SadFaceMaker` based on `FaceMaker2` that shows a sad face instead of a smiling face.

► **Exercise 5.2 A bright eyes face maker)**
Write a class called `BrightEyesFaceMaker`, based on `FaceMaker2`, that replaces the circular eyes with white elliptical ones with a blue circle inside.

► **Exercise 5.3 A happy face with ears)**
Write a class called `FaceWithEarsMaker`, based in `FaceMaker2`, that adds ears to the face as shown in Figure 5.24. Hint: If you fill and draw the ears first and then draw and fill the face it will

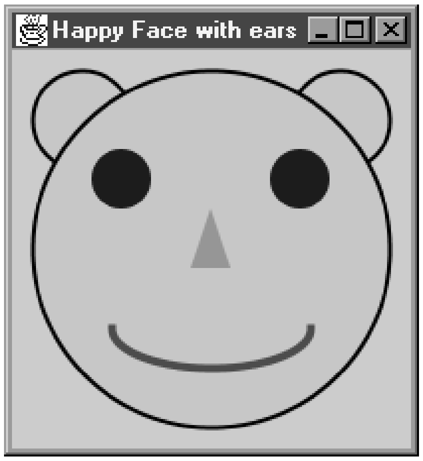


Figure 5.24: A colorful happy face with ears

cover up part of the ears.

► **Exercise 5.4 Create your own art work**
Write your own graphics program that draws an interesting picture. For example, you could draw a fish, a mouse, or a Halloween pumpkin. An example of a fish is shown in Figure 5.25. Here the fish body and tail fin are blue, the other fins are light blue, the eye is a white circle containing a black ellipse and the mouth is a red arc. Another example is shown in Figure 5.26. Another example that uses the `curveTo` method not discussed in the text is shown in Figure 5.27.

► **Exercise 5.5 (Order of draw and fill is important)**
Write a program class called `DrawFillTest` that displays two circles side by side. Use a brush width of `10.0f`, drawing color black, and fill color red. Render the left circle using `draw` followed by `fill`. Render the right circle using `fill` followed by `draw`. Explain the differences.

► **Exercise 5.6 (A pie chart)**
Write a program called `PieChart` that draws a pie chart having 5 pieces, each specified by the percentage of the entire pie that it requires. Use the percentages 4%, 10%, 11%, 15%, and 60%, which can each be converted to the number of degrees in the circle. Use a different color for each piece. Design your program so that the largest pie possible is shown centered in the window regardless of the size of the window. (Hint: the diameter of the pie is the minimum of `w` and `h`, where `w` and `h` are the width and height of the window in pixels. There is a function called `Math.min` in the `Math` class) that can find the minimum value.

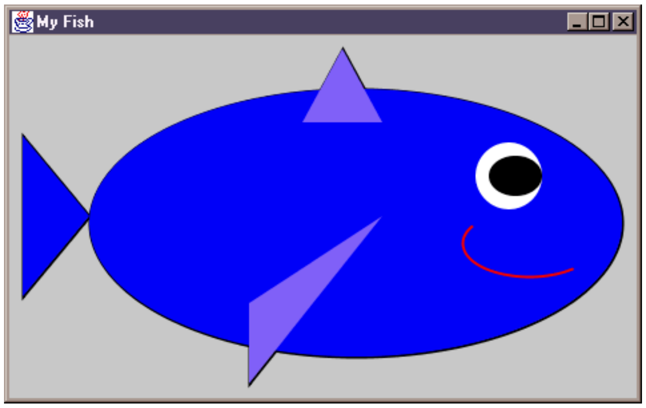


Figure 5.25: A fish

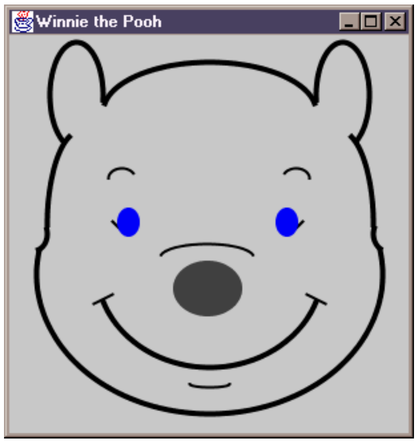


Figure 5.26: Winnie the Pooh

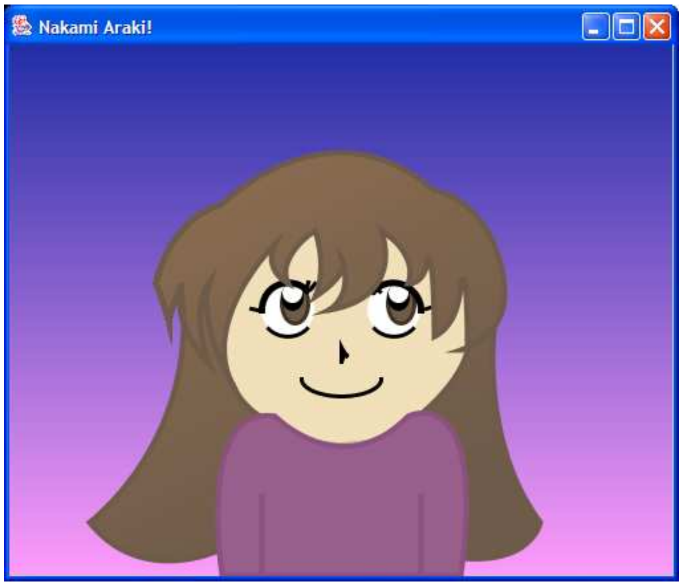


Figure 5.27: Nakami Araki

► **Exercise 5.7 (Affine transformations)**
Suppose that you have a drawing surface that is 201 pixels wide and 201 pixels high. The default user space coordinate system would have origin (0.0,0.0) at the top left corner, with bottom right corner at (200.0,200.0), and with the *y*-axis increasing from top to bottom. The following statements

```
AffineTransform at = new AffineTransform();
at.translate(100,100);
at.scale(10,-10);
g2D.setStroke(new BasicStroke(1.0f/10.0f));
g2D.transform(at);
```

will change the user space coordinate system so that the origin is in the middle of the window. The *x* coordinates now range from -10 to 10 and the *y* coordinates range from -10 to 10 increasing upwards like a normal mathematical coordinate system. Verify this by writing a program class called `AxesMaker` that uses these statements to draw the *x* and *y* axes as lines from -9 to 9 with an arrow at the end of each positive axis. The output is shown in Figure 5.28. What happens if you remove the `setStroke` statement?

► **Exercise 5.8 (Translating origin to center of window)**
The following statements can be used inside the `paintComponenet` method to change from the default user space to a user space with a right handed coordinate system whose origin is at the center of the window.

```
double xMax = getWidth() - 1;
double yMax = getHeight() - 1;
// flip the y axis
```

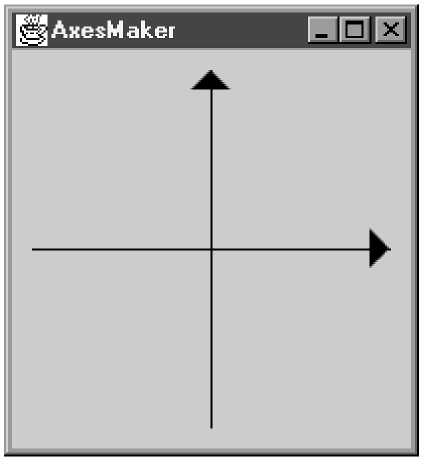


Figure 5.28: Centered coordinate system

```
AffineTransform at = new AffineTransform(1, 0, 0, -1, 0, (double) yMax);
// Translate origin to center of screen
at.translate(xMax/2, yMax/2);
g2D.transform(at);
```

Verify this by writing a program to draw the *x* axis from `-xMax/2` to `xMax/2` and the *y* axis from `-yMax/2` to `yMax/2`. Then draw a line from `(0,0)` to `(xMax/2,yMax/2)`.

▶ Exercise 5.9 (Drawing a hexagon)
Write a program called `DrawHexagon` that is similar to `DrawPentagon` except that it draws a six-sided polygon with all sides equal.

Chapter 6

Making Decisions

Conditional execution of statements

Outline

Simple boolean expressions

Relational and equality operators and expressions

Conditional operator

Comparison of floating point numbers

Simple if-else statement

If without else statement

Nested and multiple (N-way) if-statement

Common errors with if-statements

Compound boolean expressions and logical operators

Lexicographical ordering of strings using character codes

String comparison and equality

Error reporting using exceptions

Throwing and catching exceptions in `BankAccount` class

Paper, scissors, rock game

Complex roots of a quadratic equation

6.1 Introduction

In the programs we have written so far, statements are always executed one after the other in a sequential manner: the same sequence of statements is always executed. However, many algorithms are expressed in terms of conditions that can be true or false, depending on the input data for example, so the statements executed depend on these conditions. We need to be able to express the fact that one sequence of statements is to be executed if a certain condition is true and another sequence is to be executed if the condition is false.

For example, in the `TriangleCalculator` class from Chapter 3 (page 64) the constructor arguments were two sides of a triangle and the contained angle. We did not check that these values were positive. A better program would check using conditions such as $a > 0$ and $b > 0$ and only do the calculations if these conditions were true. Similarly, in Chapter 3, the `QuadraticRootFinder` class (page 67) was used to compute the roots of the quadratic equation $ax^2 + bx + c = 0$ in case both are real. The condition for real roots is $b^2 - 4ac \geq 0$ but we did not check this condition.

In this chapter we will see how the if-statement can be used for the conditional execution of statements. To express conditions we use boolean expressions, which can have true or false values, relational operators, which compare the values of arithmetic expressions to produce true or false values, and logical operators, which combine simple boolean expressions to obtain compound boolean expressions.

Boolean expressions can be tested in a program using an if-statement. When the expression has a true value one block of statements is executed and when it has a false value another block of statements is executed. This process of executing one block of statements or another, based on the value of a boolean expression, is called **conditional execution**. This will give our classes important decision making capabilities. A discussion of common if-statement errors is also included.

To illustrate these ideas we we use the the “Paper, Scissors, Rock” game. Also, we modify the `BankAccount` class, introduced in Chapter 4, to include error checking. The important concepts of an exception and throwing an exception are also introduced and illustrated using the `BankAccount` class. We also extend the `QuadraticRootFinder` class so that it finds both the real and complex roots of a quadratic equation.

6.2 Simple boolean expressions

Conditional execution is based on the evaluation of a condition. In Java the condition is a **boolean expression** which evaluates to one of the values `true` or `false` of the `boolean` data type (see Chapter 2). These two values are called **boolean literals**.

Just as there are many ways to form arithmetic expressions, the same applies to boolean expressions. The most common is to compare two arithmetic or boolean expressions using a binary comparison operator. These expressions are called **comparison expressions** and have the form

Expression1 ComparisonOperator Expression2

where *ComparisonOperator* is one of the six **comparison operators** shown in Table 6.1. This table also shows the standard mathematical notation for these operators. A double equal sign represents equality in Java since the single equal sign is already used for assignment. The two expressions are evaluated first before the comparison operator is applied.

Comparison Operator	**Mathematical Notation**	**Meaning**
`>`	$>$	greater than
`>=`	$\geq$	greater than or equal
`<`	$<$	less than
`<=`	$\leq$	less than or equal
`==`	$=$	equal
`!=`	$\neq$	not equal

Table 6.1: The six comparison operators

The six comparison operators fall into two groups: `==` and `!=` are called **equality operators**, since they test expressions for equality or inequality, and the other four operators are called **relational operators**. The corresponding expressions have the form

ArithmeticExpression1 RelationalOperator ArithmeticExpression2

for **relational expressions**, and the form

Expression1 EqualityOperator Expression2

for **equality expressions**, where *Expression1* and *Expression2* can be either boolean or arithmetic expressions.

■ EXAMPLE 6.1 **(Simple boolean expressions)**

(a) The expression `month == 3` is true only if `month` has the value 3.

(b) The expression `k % 2 == 0` is true only if the integer `k` is even, which is the case if the remainder on division by 2 is 0. The expression `k % 2 != 0` is true only if the integer `k` is odd. Generalizing, the expression `k % n == 0` is true only if `k` is divisible by the integer `n` (remainder on division by n is 0).

(c) The expression `b*b - 4.0*a*c >= 0` is true only if the quadratic equation $ax^2 + bx + c = 0$ has real roots. For example it is false if a, b, and c all have the value 1, and true if $a = 1$, $b = 3$, and $c = 2$.

(d) If `playerChoice` is a variable of type `char`, the expression `playerChoice == 'P'` is true only if the variable has the value `'P'`.

(e) In a turtle graphics system the turtle has a pen that can be either up or down. This condition can be represented by a variable called `penUp` of type `boolean`, where a value of `true` indicates that the pen is up. A boolean variable such as `penUp` is a simple example of a boolean expression since it evaluates to a `true` or `false` value.

In examples (a) to (d) the expressions on either side of the comparison operator are evaluated before the comparison operator is applied. ■

```
if ( BooleanExpression )
{
   Statements A
}
else
{
   Statements B
}
```

Figure 6.1: A template for the if-statement

6.3 If-statements

Conditional execution can be accomplished using an if-statement Since the syntax varies with the computer language it is useful to express it using the following language independent **algorithmic notation** often called **pseudo-code**.

```
IF BooleanExpression THEN
   Statements A
ELSE
   Statements B
END IF
```

Here *BooleanExpression* stands for any expression that evaluates to one of the values true or false. The statements labeled *A* are executed if *BooleanExpression* is true and the statements labeled *B* are executed if it is false. In Java the corresponding if-statement has the structure shown in Figure 6.1. The parentheses enclosing *BooleanExpression* are necessary. A sequence of statements enclosed in braces is called a **block** so the if-statement defines two blocks, one for each value of *BooleanExpression*. The first block is called the **if-block** and the second block is called the **else-block**.

The statements in the two blocks are indented by an equal amount of space. Indentation has no effect on execution. It is there to improve the readability. We recommend using three spaces of indentation for the statements in a block.

The template in Figure 6.1 is a static diagram, designed to show the syntax and layout to use when writing if-statements. It does not show the flow of execution. A **flowchart** is a graphical representation of the flow of execution. The flowchart for the if-statement is shown in Figure 6.2. The downward arrow at the top indicates the flow before the if-statement is encountered. Then the diamond-shaped box represents the boolean expression to be evaluated. One of the outward arrows is chosen depending on the value of the expression. Rectangular boxes contain statements to be executed sequentially. To follow the flow, begin at the top and follow the arrows until you reach the bottom. In any case exactly one of the two blocks *A* and *B* will be executed. The downward arrow at the bottom represents the flow after the if-statement.

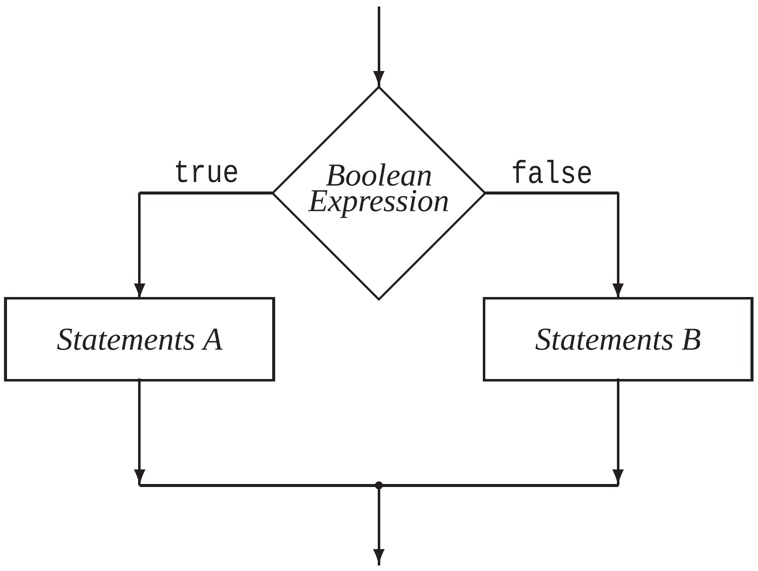


Figure 6.2: A flowchart for the execution of an if-statement

■ EXAMPLE 6.2 **(Calculating the absolute value)** The absolute value $|x|$ of x is defined to be x if $x \geq 0$ and $-x$ if $x < 0$. It can be calculated using `Math.abs`. If we didn't have this function in the `Math` class we could use the following method

```
double abs(double x)
{
   if (x >= 0)
   {
      return x;
   }
   else
   {
      return -x;
   }
}
```

which returns the absolute value of a `double` number. ■

■ EXAMPLE 6.3 **(A cube root method)** If x is a real number then its cube root $x^{1/3}$ can be calculated using `Math.pow(x, 1.0/3.0)` but only if $x \geq 0$. If $x < 0$ we can write $x^{1/3} = -(-x)^{1/3}$ and use `-Math.pow(-x, 1/.0/3.0)`. The method

```
double cubeRoot(double x)
{
   if (x >= 0)
   {
      return Math.pow(x, 1.0/3.0);
   }
   else
   {
      return -Math.pow(-x, 1.0/3.0);
   }
}
```

uses an if-statement to return the cube root of x in either case. ■

6.4 Real roots of a quadratic equation

In Chapter 3 we wrote a `QuadraticRootFinder` class (page 67) to find the real roots of a quadratic equation. We can now modify it to determine if there are real roots. To do this we add the data field

```
    private boolean realRoots;
```

This variable will be set to true if the equation has real roots and false otherwise. The “get” method

```
    public boolean hasRealRoots()
    {
       return realRoots;
    }
```

can be used to determine if the equation has real roots. We need to modify the `doCalculations` method to use an if-statement to give a value to this boolean variable depending on the sign of $b^2 - 4ac$.

6.4.1 `QuadraticRootFinder` class

Here is the complete class with these modifications.

Class `QuadraticRootFinder`

book-projects/chapter6/root_finder

```
package chapter6.root_finder; // remove this line if you’re not using packages
/**
 * An object of this class can calculate the real roots of the
 * quadratic equation ax^2 + bx + c = 0 given the coefficients a, b, and c.
 * In this version there is a check for real roots.
 */
public class QuadraticRootFinder
{
   private double a, b, c;
   private double root1, root2;
   private boolean realRoots;
   /**
    * Construct a quadratic equation root finder given the coefficients
    * @param a first coefficient in axˆ2 + bx + c
    * @param b second coefficient in axˆ2 + bx + c
    * @param c third coefficient of axˆ2 + bx + c
    */
   public QuadraticRootFinder(double aCoeff, double bCoeff, double cCoeff)
   {
      a = aCoeff;
      b = bCoeff;
      c = cCoeff;
      doCalculations();
   }

   private void doCalculations()
   {
      double d1 = b*b - 4*a*c;
      if (d1 >= 0)
      {
         double d = Math.sqrt(d1);
         root1 = (-b - d) / (2.0 * a);
         root2 = (-b + d) / (2.0 * a);
         realRoots = true;
      }
      else
      {
         realRoots = false;
      }
   }

   /**
    * Returns true if real roots were found else false.
    * @return true if real roots were found else false
    */
   public boolean hasRealRoots()
   {
      return realRoots;
   }

   // getRoot1 and getRoot2 methods from Chapter 3 go here

   // getA, getB, and getC methods from Chapter 3 go here

   // setA, setB, and setC methods from Chapter 3 go here

}
```

6.5 Block declaration of variables

A **block** is any sequence of statements delimited by braces. Variables are defined in blocks and are said to have **block scope**. This means that they do not exist outside the block in which they are declared. We have now seen three kinds of blocks:

- Data fields have the widest scope. They are defined in the class declaration block so they are available anywhere in the class. The variables `a`, `b`, and `c` in the `QuadraticRootFinder` class are examples.
- Local variables in a constructor or method are defined only in the block defining the constructor or method body. The variable `d1` defined in the `doCalculations` method is an example.
- Variables declared in the if-block or the else-block of an if-statement are local to this block. The variable `d` defined in the if-block of the `doCalculations` method is an example.

6.6 If-statement with no else

When the else-part of the if-statement is not required, it can be omitted to give the pseudo-code statement

```
IF BooleanExpression THEN
    Statements
END IF
```

or the Java statement

```
if (BooleanExpression)
{
   // statements
}
```

The if-block is executed only if the boolean expression is true. Otherwise it is skipped and execution resumes with any statements after the if-statement. The flowchart for the if-statement with no else-part is shown in Figure 6.3.

■ EXAMPLE 6.4 **(If-statement with no else-part)** In `QuadraticRootFinder` we could have written the `doCalculations method as`

```
private void doCalculations()
{
   realRoots = false;
   double d1 = b*b - 4*a*c;
   if (d1 >= 0)
   {
      double d = Math.sqrt(d1);
      root1 = (-b - d) / (2.0 * a);
      root2 = (-b + d) / (2.0 * a);
      realRoots = true;
   }
}
```

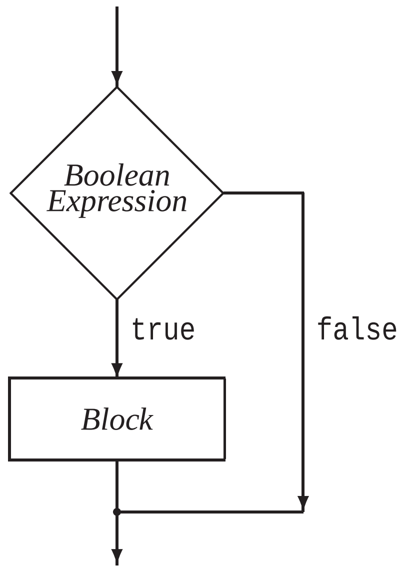


Figure 6.3: Flowchart for an if-statement with no else-part

which initializes `realRoots` to `false` so no else-part is required. ■

■ EXAMPLE 6.5 **(Assigning boolean expressions)** Another variation of Example 6.4 is

```
private void doCalculations()
{
   double d1 = b*b - 4*a*c;
   realRoots = d1 >= 0;
   if (realRoots)
   {
      double d = Math.sqrt(d1);
      root1 = (-b - d) / (2.0*a);
      root2 = (-b + d) / (2.0*a);
   }
}
```

which uses a boolean assignment statement to assign the true or false value of the boolean expression `d1 >= 0` directly to the variable `realRoots`, which now becomes the condition in the if-statement. ■

■ EXAMPLE 6.6 **(One line if-statement)** The following statements compute the maximum of the integer variables `x` and `y` and assign it to `max`.

```
int max = x; // assume x is the maximum
if (y > max)
{
   max = y; // replace max with y if y is bigger than x
}
```

Since there is only a single statement in the if-block the braces can be omitted so you will often see the if-statement written as

```
if (y > max)
   max = y;
```

or even as the so-called one line if-statement

```
if (y > max) max = y;
```

The same rule applies to the else-block. Another variation of the maximum calculation is

```
int max;
if (x > y)
   max = x;
else
   max = y;
```

If you find the use of braces more readable always use them. Several common errors can occur by not using braces (see Section 6.10). ■

6.7 Comparison of floating point numbers

Most floating point numbers cannot be stored exactly in computer memory. For example, the simple decimal number 0.1 cannot be stored exactly because its binary value has an infinite number of digits. This leads to truncation error. Computers also have a limited accuracy when performing computations. Each arithmetic operation may introduce a small roundoff error, and as more operations are carried out, roundoff error can accumulate. In practice we may consider the two numbers 1 and 0.99999999 to be “equal”, but they may not be equal to the computer. Therefore, it is advisable not to compare two expressions of type `double` directly, using the comparison operators, especially the equality operators. Instead, either the **absolute error** or the **relative error** can be used.

6.7.1 Floating point tester class

Errors can be demonstrated using the following class that computes π^5 in two ways, once using `Math.pow`, and the other using four multiplications. The class can be run both inside and outside BlueJ.

Class `FloatingPointTester1`

book-projects/chapter6/floating_point

```
package chapter6.floating_point; // remove this line if you’re not using packages
/**
 * Testing equality of floating point numbers using equality.
 */
```

```
public class FloatingPointTester1
{
   public void doTest()
   {
      double x = Math.pow(Math.PI, 5.0);
      double y = Math.PI * Math.PI * Math.PI * Math.PI * Math.PI;
      System.out.println("1st approx is " + x);
      System.out.println("2nd approx is " + y);

      if (x == y)
         System.out.println("equal");
      else
         System.out.println("not equal");
   }

   public static void main(String[] args)
   {
      FloatingPointTester1 tester = new FloatingPointTester1();
      tester.doTest();
   }
}
```

■ $\boxed{\text{EXAMPLE 6.7}}$ **(Using == to compare floating point numbers)** The `FloatingPointTester1` class produces the output

```
1st approx is 306.0196847852814
2nd approx is 306.01968478528136
not equal
```

The `pow` method actually uses the formula $e^{5\ln\pi} = \pi^5$ to compute its result rather than direct multiplication. The comparison finds them to be "not equal" since there will be differing amounts of round-off error in the two calculations. In fact `x` is slightly greater than `y`. This example shows that we must be careful when comparing floating point numbers directly. ■

■ $\boxed{\text{EXAMPLE 6.8}}$ **(Absolute error for floating point comparison)** If you want to compare two arithmetic expressions of type `double` for equality, or inequality, it may be better to use

$$\text{absoluteError} = |x - y|$$

as a measure of equality. This defines the absolute error between the two values x and y as the absolute value of the difference between the two numbers. We can use it to check if the absolute difference between two numbers is smaller than a very small number, say `1E-10`. Replace the if-statement in `FloatingPointTester1` with

```
if (Math.abs(x - y) <= 1E-10)
   System.out.println("equal");
else
   System.out.println("not equal");
```

to get `FloatingPointTester2` and the "equal" message will be printed. This means that two floating point numbers should be considered equal if they are close enough to each other as defined by `1E-10`. ■

■ EXAMPLE 6.9 **(Relative error for floating point comparison)** In scientific calculations the relative error is often a better measure of the closeness of two floating point numbers x and y. If $x \neq 0$, one definition is

$$\text{relativeError} = \frac{x - y}{x}$$

We can replace the if-statement in `FloatingPointTester2` with

```
double relativeError = (x - y) / x;
if (Math.abs(relativeError) <= 1E-10)
   System.out.println("equal");
else
   System.out.println("not equal");
```

to get `FloatingPointTester3` and the "equal" message will be printed. ■

6.8 Conditional operator

There is a special operator in Java called the **conditional operator**, denoted by `?:`, which can be used to write special if-statements in a compact fashion. It produces conditional expressions of the form

booleanExpression `?` *expressionA* `:` *expressionB*

The value of the conditional expression is *expressionA* if *booleanExpression* is `true` and *expressionB* otherwise. The value of the conditional expression can then be assigned to a variable. For example, if *expressionA* and *expressionB* evaluate to an `int` value then we can write a statement such as

`int v =` *booleanExpression* `?` *expressionA* `:` *expressionB* `;`

The conditional operator is not really necessary since you can achieve the same result with

```
int v;
if ( booleanExpression )
   v =  expressionA ;
else
   v =  expressionB ;
```

We will not use the conditional operator much since it can make programs harder to read.

■ EXAMPLE 6.10 **(The conditional operator)** The absolute value method in Example 6.2 can also be defined as

```
double abs(double x)
{
   return (x >= 0) ? x : -x;
}
```

using the conditional operator. ■

■ EXAMPLE 6.11 **(The conditional operator)** In Example 6.6 the maximum of two integer variables `x` and `y` was computed using an if-statement. This can also be done as

```
int max = (x >= y) ? x : y;
```

using the conditional operator. ■

■ EXAMPLE 6.12 **(The conditional operator version of cubeRoot)** The `cubeRoot` method from Example 6.3 can be expressed as

```
double cubeRoot(double x)
{
   return (x >= 0) ? Math.pow(x, 1.0/3.0) : -Math.pow(-x, 1.0/3.0);
}
```

using the conditional operator. ■

6.9 Nested and multiple (N-way) if-statements

In the general if-statement in Figure 6.1 the two blocks can also contain other if-statements. If-statements within if-statements are said to be nested.

■ EXAMPLE 6.13 **(A nested if-statement)** Suppose we have an amount of money $a \geq 0$ and the tax is 5% if $0 \leq a < 10000$, 10% if $10000 \leq a < 100000$, and 15% if $a \geq 100000$. The nested if-statement

```
double tax;
if (a >= 10000)
{
   if (a < 100000)
   {
      tax = 0.10 * a;
   }
   else // a >= 100000
   {
      tax = 0.15 * a;
   }
```

```
if ( BooleanExpression1 )
{
   Statements 1
}
else if ( BooleanExpression2 )
{
   Statements 2
}
...
else if ( BooleanExpressionN )
{
   Statements N
}
else
{
   Default statements
}
```

Figure 6.4: Template for the multiple if-statement

```
}
else // a < 10000
{
   tax = 0.05 * a;
}
```

shows one way to compute the tax using a nested if-statement inside the outer if-block. We have included comments on the else parts to improve readability. ■

The if-else-statement is designed for a two-way decision process. It can be generalized to a multiple if-statement, sometimes called an if-else-if-statement, that is a special kind of nested if-statement designed for a multi-way decision process. The template is a generalization of the one given in Figure 6.1 and is shown in Figure 6.4.

There are N conditions, represented by N boolean expressions, and $N+1$ blocks. The N conditions do not have to be mutually exclusive. There may be more than one condition that evaluates to true. But in this case only the block for the first of these conditions will be executed. The order of the tests is important in this case.

If the N conditions are mutually exclusive, meaning that only one of them at a time can be true, then the order of the conditions is not important. In any case, exactly one of the $N+1$ blocks of statements is executed. If none of the N conditions is true, the default block is executed. A flowchart for the multiple if-statement is shown in Figure 6.5.

■ EXAMPLE 6.14 **(Calculating letter grades)** The following if-statement assigns a letter grade for a given integer mark:

```
String letterGrade;
if (mark < 0)
   letterGrade = "";
else if (mark > 100)
   letterGrade = "";
else if (mark >= 80)
   letterGrade = "A";
else if (mark >= 70)
   letterGrade = "B";
else if (mark >= 60)
   letterGrade = "C";
else if (mark >= 50)
   letterGrade = "D";
else
   letterGrade = "F";
```

Here marks outside the range 0 to 100 are assigned an empty string as a letter grade. The order of the conditions is important here since they are not mutually exclusive. For example, using `marks >= 50` first will not work since any mark of 50 or more will result in a grade of D being assigned. ■

6.10 Common errors with if-statements

The following three examples show errors that can occur if you are not careful with braces. They can be avoided by always using braces.

■ EXAMPLE 6.15 **(Forgetting the braces)** Consider the if-statement in Figure 6.6(a) The intent here is to assign the maximum of `x` and `y` to `max` and the minimum to `min`. However, because there are no braces in the else-block the compiler assumes that only the assignment to `max` belongs to this block. The indentation is misleading and has no effect on the compiler. Thus, `min` will always receive the value of `x` since this assignment statement is not part of the if-else statement. This is an example of a logical error. The if-statement in Figure 6.6(b) corrects the problem using braces. ■

■ EXAMPLE 6.16 **(Else without if)** If you forget to use braces for the if-block, as in the if-statement in Figure 6.6(c) the Java compiler will now report an “else without if” error message. Since there are no braces in the if-block the compiler assumes that

```
if (x >= y)
   max = x;
```

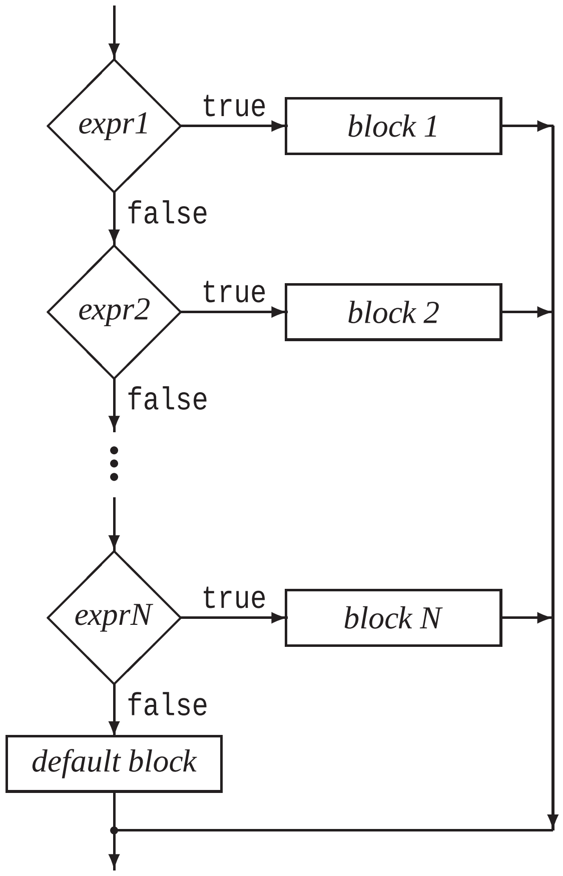


Figure 6.5: A flowchart for the execution of a multiple if-statement

```
if (x >= y)
{
   max = x;
   min = y;
}
else
   max = y;
   min = x;
```

(a)

```
if (x >= y)
{
   max = x;
   min = y;
}
else
{
   max = y;
   min = x;
}
```

(b)

```
if (x >= y)
   max = x;
   min = y;
else
{
   max = y;
   min = x;
}
```

(c)

Figure 6.6: forgetting the braces

is a complete if-statement with no else part. Then the statement `min = y` is a normal statement not inside an if-statement. Then the `else` is encountered with no matching `if`, and this is a syntax error. ■

■ EXAMPLE 6.17 **(Dangling else problem)** Consider the following statement with a nested if-statement:

```
if (mark >= 50)
   if (mark <= 100)
      System.out.println("Pass");
else
   System.out.println("Fail");
```

The intent here is to display `Pass` in case the mark is in the range 50 to 100 and `Fail` otherwise. However, for marks less than 50 nothing is ever displayed and for marks greater than 100 the `Fail` message is displayed.

Again the indentation is misleading since it seems to associate the `else` part with the outer `if`. However, it is also possible to associate the `else` with the inner `if` and the results are not the same. This is called the "dangling else problem".

To resolve this ambiguity the compiler always associates an `else` with the nearest `if` so the if-statement is interpreted as

```
if (mark >= 50)
{
   if (mark <= 100)
      System.out.println("Pass");
   else
      System.out.println("Fail");
}
```

To obtain the desired meaning, and associate the `else` with the outer `if`, we need to use braces:

```
if (mark >= 50)
{
   if (mark <= 100)
      System.out.println("Pass");
}
else
{
    System.out.println("Fail");
}
```

which now displays the `Fail` message for marks less than 50. ■

Java Notation	Mathematical Notation	Meaning
`&&`	$\wedge$	logical “and” (looks like an “A”)
`||`	$\vee$	logical “or” (looks like an “r”)
`!`	$\sim$, $\neg$	logical “not” (negation)

Table 6.2: The three basic logical operators

6.11 Compound boolean expressions

A boolean expression is an expression which can have a true or false value. The simplest boolean expression is just a boolean variable itself, for example `realRoots` in Example 6.5. Boolean expressions can also be obtained using the comparison operators in Table 6.1. A **compound boolean expression** consists of two or more boolean expressions connected together by one or more logical operators. These operators can be used to express conditions that can be true in more than one way.

The mathematical and Java notations for the three basic logical operators are shown in Table 6.2. In mathematics, $\wedge$ is often called “wedge” and $\vee$ is often called “vee”. There are several mathematical notations for negation. Two are shown in the table, namely $\sim$, which is called “tilde”, and $\neg$. Both are called “not”. We will use $\sim$. In pseudo-code you can use either the mathematical notation or the names AND, OR, and NOT.

The symbols $\wedge$ and $\vee$ are not available on keyboards so Java uses `&&` and `||` instead[1]. For negation the exclamation mark is used.

6.11.1 Writing expressions using AND, OR, and NOT

If b_1, b_2, ..., b_n are n boolean expressions then

$b_1 \wedge b_2 \wedge \ldots \wedge b_n$ (`b1 && b2 && ... && bn`, in Java)

is the compound expression obtained by “and”ing together the n expressions. It is called the “logical and”. The value of this compound expression is true only if all n expressions b_1 to b_n are true. If any expression is false then the compound expression is false. A **truth table** gives the values of a compound boolean expression in terms of all possible values of the simple boolean expressions it contains. In the case of two simple expressions, p and q, the truth table for $p \wedge q$ is shown in Figure 6.3(a). For example, the first row of this table tells us that if p is false and q is false, then $p \wedge q$ is also false. The last column of the table clearly shows that $p \wedge q$ is true only if both p and q are true.

Similarly, we can perform the $\vee$ operation and

$b_1 \vee b_2 \vee \ldots \vee b_n$ (`b1 || b2 || ... || bn`, in Java)

is the compound expression obtained by “or”ing together the n expressions. It is called the “logical or”. The value of this compound expression is true if any one of the n expressions b_1 to b_n is true.

[1]Double symbols are used since `&` and `|` have other meanings in Java which we will not discuss (bitwise operators).

p	q	$p \wedge q$
false	false	false
false	true	false
true	false	false
true	true	true

(a)

p	q	$p \vee q$
false	false	false
false	true	true
true	false	true
true	true	true

(b)

p	$\sim p$
false	true
true	false

(c)

Table 6.3: Truth tables for AND, OR, and NOT

It is false only if all n expressions are false. In the case of two simple expressions p and q the truth table for $p \vee q$ is shown in Figure 6.3(b). The last column of the table clearly shows that $p \vee q$ is false only if both p and q are false.

The operators $\wedge$ and $\vee$ are binary operators because they operate on two boolean expressions. The $\sim$ operator is an example of a unary operator. It operates on only one boolean expression to give the opposite truth value (negation) of the expression. The truth table for $\sim p$ is shown in Figure 6.3(c).

Operator precedence rules

Among the unary operators `-`, `+`, `!`, the binary arithmetic operators `*`, `/`, `%`, `+`, `-`, the comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=`, and the binary logical operators `&&`, `||`, we have the following precedence rules in the order from highest to lowest:

1. Parentheses `( )` have the highest precedence.
2. The unary operators `-`, `+`, and the unary negation operator `!`, have equal precedence and they are right associative (they are applied in right to left order).
3. The binary arithmetic operators `*`, `/`, and `%` have equal precedence and they are left associative (they are applied in left to right order).
4. The binary arithmetic operators `+` and `-` have equal precedence and they are left associative.
5. The relational operators `<`, `<=`, `>`, and `>=` have equal precedence and are not associative.
6. The equality operators `==` and `!=` have equal precedence and are left associative.
7. The binary logical "and" operator `&&` is left associative.
8. The binary logical "or" operator `||` is left associative.

■ EXAMPLE 6.18 **(Applying precedence rules)** In expression `mark >= 0 && mark <= 100` the expressions `mark >= 0` and `mark <= 100` are evaluated first and then `&&` is applied. To emphasize this the expression could be written using parentheses as `(mark >= 0) && (mark <= 100)`.

Assuming that `d` and `p` are of type `int`, the compound expression `! d == 0 || p == 0` results in the compiler error "can't convert int to boolean". The `!` operator is being applied to `d` since it has a higher precedence than `==`. Parentheses are needed to obtain `!(d == 0) || p == 0`. Now `d == 0` is evaluated and then `!` is applied. Then `p == 0` is evaluated and finally `||` is applied. ■

■ EXAMPLE 6.19 **(Mixed boolean expressions)** The $\wedge$ and $\vee$ operators can be mixed together. In the expression $a \wedge b \vee c$ the expression $a \wedge b$ is evaluated first according to the precedence rules, then $\vee$ is applied. To evaluate $b \vee c$ first it is necessary to use parentheses and write $a \wedge (b \vee c)$. ■

■ EXAMPLE 6.20 **(Truth table for $(a \wedge b) \vee c$ and $a \wedge (b \vee c)$)** The truth table requires 8 rows, since there are $2^3 = 8$ possible combinations for the values of a, b, and c. This gives the truth table

a	b	c	$a \wedge b$	$b \vee c$	$(a \wedge b) \vee c$	$a \wedge (b \vee c)$
false	false	false	false	false	false	false
false	false	true	false	true	true	false
false	true	false	false	true	false	false
false	true	true	false	true	true	false
true	false	false	false	false	false	false
true	false	true	false	true	true	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

where the results for $(a \wedge b) \vee c$ and $a \wedge (b \vee c)$ in the final two columns show that the expressions are not the same. ■

DeMorgan's laws

If a and b are boolean expressions, we have the laws

$$\begin{aligned} \sim(a \wedge b) &= (\sim a) \vee (\sim b) \\ \sim(a \vee b) &= (\sim a) \wedge (\sim b) \end{aligned}$$

for negating compound expressions. These laws are called deMorgan's laws and can be easily verified using truth tables. The generalizations to n expressions b_1, b_2, ..., b_n are

$$\begin{aligned} \sim(b_1 \wedge b_2 \wedge \ldots \wedge b_n) &= (\sim b_1) \vee (\sim b_2) \vee \ldots \vee (\sim b_n) \\ \sim(b_1 \vee b_2 \vee \ldots \vee b_n) &= (\sim b_1) \wedge (\sim b_2) \wedge \ldots \wedge (\sim b_n) \end{aligned}$$

Testing numerical ranges

Often it is necessary to test if a variable n has a value in the range a to b. In mathematics this is expressed as $a \le n \le b$. In programming languages we cannot use the expression `a <= n <= b`. Instead, it is expressed as `a <= n && n <= b` using `&&` to connect two relational expressions.

■ EXAMPLE 6.21 **(Numerical ranges)** In Example 6.14 a letter grade was assigned to a mark using a multiple if-statement. The conditions were not mutually exclusive so the order was important. Using compound logical expressions involving &&, we can make the conditions mutually exclusive as follows:

```
if (80 <= mark && mark <= 100)
   letterGrade = "A";
else if (70 <= mark && mark < 80)
   letterGrade = "B";
else if (60 <= mark && mark < 70)
   letterGrade = "C";
else if (50 <= mark && mark < 60)
   letterGrade = "D";
else if (0 <= mark && mark < 50)
   letterGrade = "F";
else
   letterGrade = ""; // invalid mark case
```

Now the order of the conditions is not important. The else-block is used to trap errors. When possible it is best to write the conditions in a multiple if-statement in mutually exclusive form, since these conditions are easier to understand, and use the else-block to trap invalid cases, ■

■ EXAMPLE 6.22 **(Numerical ranges)** Using compound boolean expressions we can rewrite Example 6.13 as

```
double tax;
if (0 <= a && a < 10000)
{
   tax = 0.05 * a;
}
else if (10000 <= a && a <= 100000)
{
   tax = 0.10 * a;
}
else // a >= 100000
{
   tax = 0.15 * a;
}
```

which is much easier to read. ■

6.11.2 Leap year problem

In a leap year February has 29 days instead of 28. There are two statements that define when a year is a leap year:

s_1 = the year is divisible by 4 but not by 100 (for example, 1988 or 1992)

s_2 = the year is divisible by 400 (for example, 1600 or 2000)

If either of these statements is true then the year is a leap year, so if we let s be the statement “the year is a leap year” then s is the compound statement

$$s \quad = \quad \text{the year is a leap year} \; = \; s_1 \vee s_2$$

We need to translate s_1 and s_2 into logical expressions. They contain the following three simpler logical expressions:

$$\begin{aligned} a &= \text{year is divisible by 4} \\ b &= \text{year is not divisible by 100} \\ c &= \text{year is divisible by 400} \end{aligned}$$

Therefore, we can write (interpreting “but” as “and”)

$$\begin{aligned} s_1 &= a \wedge b, \\ s_2 &= c, \\ s &= s_1 \vee s_2 = (a \wedge b) \vee c. \end{aligned}$$

Using these expressions we can write the pseudo-code algorithm shown in Figure 6.7. In this

```
**ALGORITHM** LeapYear(*year*)
$a \leftarrow$ *year* is divisible by 4
$b \leftarrow$ *year* is not divisible by 100
$c \leftarrow$ *year* is divisible by 400
IF $(a \wedge b) \vee c$ THEN
    RETURN true (year is a leap year)
ELSE
    RETURN false (year is not a leap year)
END IF
```

Figure 6.7: Pseudo-code algorithm for the leap year problem

pseudo-code algorithm we use the left arrow symbol to indicate assignment. In Java, if `year` is an integer variable then

a translates to `year % 4 == 0`
b translates to `year % 100 != 0`
c translates to `year % 400 == 0`

and we obtain the boolean leap year expression

```
(year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
```

The && operator will be performed first, since it has a higher precedence than `||`.

■ EXAMPLE 6.23 **(Leap year test)** To test the leap year expression try the following statements in the BeanShell workspace editor (select it from the File menu).

```
int year = 2004;
if ( (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0))
   print(year + " is a leap year");
```

From the editor's "Evaluate" menu select "Eval in workspace" and you will see the message "2004 is a leap year" in the workspace. Try other years and select "Eval in workspace" each time. ■

6.11.3 Short circuit evaluation

In Java the boolean expression `p && q` is evaluated using what is called **short circuit evaluation**. The idea is that if `p` is evaluated and it is `false`, the entire expression will be `false`, so there is no need to evaluate `q`. However, if `p` is `true` then it is necessary to evaluate `q` to obtain the truth value of `p && q`. A similar rule is used for `p || q`. Here `p` is evaluated first and if it has the value `true` the entire expression is `true` so there is no need to evaluate `q`. In Java the order of expressions in the compound boolean expressions $p \wedge q$ and $p \vee q$ may be important even though in mathematics the order is unimportant since $p \wedge q = q \wedge p$ and $p \vee q = q \vee p$.

■ EXAMPLE 6.24 **(Short-circuit evaluation)** If `x` and `y` are variables of type `int` the if-statement

```
if (x != 0 && y/x > 2) {...}
```

does not result in a division by zero when `x` is zero. The expression `x != 0` will be evaluated first to `false` in this case, and the second expression will not be attempted. However, when the order of the expressions is reversed, as in the if-statement

```
if (y/x > 2 && x != 0) {...}
```

an error will result when `x` happens to be zero. ■

6.12 String comparison and equality

The `String` class was introduced in Chapter 4. We now consider string comparison methods. In Table 6.1 we introduced the four relational operators `<`, `<=`, `>=`, and `>` comparing two arithmetic expressions, and the two equality operators, `==` and `!=`. We cannot use these operators to compare two strings. In particular the `==` operator compares two string references for equality not the string objects that they reference.

■ EXAMPLE 6.25 **(Incorrect use of == for string comparison)** To test this try the following statements in the BeanShell workspace editor (select it from the File menu).

```
String s1 = "hello";
String s2 = "hello";
if (s1 == s2)
   print("equal");
else
   print("not-equal");
```

and evaluate it in the work space. The result “not-equal” will be displayed no matter what strings are used. The reason is that the variables `s1` and `s2` are not `String` objects. They are references to `String` objects and will always have different values, even if they reference the same string object (see Part (b) of Figure 4.13). It is the references that are being compared by the `==` operator not the characters in the string objects. ■

6.12.1 Equals method for string omparison

In order to compare `String` objects themselves for equality, the `String` class has an instance method called `equals` with the prototype

```
public boolean equals(Object obj)
```

The argument is an `Object` type rather than a `String` type for technical reasons that we will discuss in Chapter 10. when we study inheritance and polymorphism. An object of any class is of type `Object` and in particular a string is also of type `Object` so we can use a string as an argument. For now just assume that the argument is of type `String`.

The prototype tells us that we can send the `equals` message to a string asking if it is equal to the argument string. The result will be true if the two string objects are equal, which means they have the same length and they contain the same characters. The comparison is case sensitive so the string `"abc"` is different from the string `"aBc"`.

■ EXAMPLE 6.26 **(String equality using equals method)** If we change Example 6.25 to use `equals` to obtain

```
String s1 = "hello";
String s2 = "hello";
if (s1.equals(s2))
   print("equal");
else
   print("not-equal");
```

then the if-statement works as expected. To test for inequality you can use the negation in an if-statement of the form

```
if (! s1.equals(s2)) {...}
```

Here `s1.equals(s2)` is evaluated and `!` is applied to negate the result. ■

■ EXAMPLE 6.27 **(Selecting menu choices)** In an interactive program you may display a menu of choices and then ask the user to select one of the choices. Suppose the choices are the strings `"addition"`, `"subtraction"`, `"multiplication"`, or `"division"`. The statements

```
if (choice.equals("addition"))
   // process addition choice here
else if (choice.equals("subtraction"))
   // process subtraction choice here
```

```
     else if (choice.equals("multiplication"))
        // process multiplication choice here
     else if (choice.equals("division"))
        // process division choice here
     else
        // process invalid choice here
```

can be used to process the different choices. ■

6.12.2 Lexicographical ordering of strings

Sometimes we want to check two strings not to see if they are equal but to determine which one comes first in the **lexicographical ordering** of strings. This ordering is defined by comparing characters from the strings one at a time.

Each character in Java is internally represented by a 16-bit integer, called a **character code**, using the Unicode system. The usual North American subset for the English language (punctuation, digits `'0'` to `'9'`, uppercase letters `'A'` to `'Z'`, and lowercase letters `'a'` to `'z'`), occupy the first 128 positions in this 16-bit code. This 128 character subset is called the ASCII code. The next 128 codes contain various accented characters, used by languages such as German and French, and the first 256 Unicode characters form what is called the ISO-LATIN1 character set.

The following simple class can be used to find out the codes corresponding to various characters.

Class `CharacterDecoder`

book-projects/chapter6/strings

```
package chapter6.strings; // remove this line if you're not using packages
/**
 * Finding the code for a given character
 */
public class CharacterDecoder
{
   /**
    * Return integer code of a character
    * @param c the character to decode
    * @return the integer code of c
    */
   public int code(char c)
   {
      int code = (int) c;
      return code;
   }
}
```

Here a typecast is used to convert the character `ch` to an integer value. The results for the printable characters in the range 0 to 127 are shown in Table 6.4. To find the ASCII code for a character in the table, add the number at the beginning of its row to the number at the top of its column. For

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Table 6.4: ASCII codes for characters

example, the code for `'Z'` is $80 + 10 = 90$. The first 32 codes (0 to 31) are not shown, since they are non-printable control codes. For example, carriage return has code 13 and is denoted by `'\r'`, line feed has code 10 and is denoted by `'\n'`, and the tab character has code 9 and is denoted by `'\t'`.

In this code the digits `'0'` to `'9'` have smaller codes than uppercase letters, which in turn have smaller codes than the lowercase letters. The codes for `'0'` to `'9'` are increasing, the codes for `'A'` to `'Z'` are increasing, and the codes for `'a'` to `'z'` are increasing. This means that we can apply the six comparison operators to characters in a relational expression and Java will use their codes. This imposes an ordering on the character set so we can say, for example, that `'a' < 'f'`, `'A' < 'C'`, `'A' < 'a'`, `'5' < 'A'`, `'5' < '6'`, and so on.

Using character codes to order strings

A string ordering can now be defined using the ordering of the string characters based on their character codes. This is called the lexicographical order. You start comparing the two strings one character at a time until either of the following conditions are true:

1. the character in one string is different from the corresponding one in the other string
2. one of the strings ends before the other

In case 1 the string whose character has the smaller code at the first position where they differ is said to precede the other (is "smaller than" the other) in the lexicographical ordering. For example, "Frank" precedes "Fred", since the "a" at index 2 of "Frank" has a smaller code than the "e" at index 2 of "Fred". As another example, "Bobby" comes after "Bob". Their first three characters match but then there are no more characters in "Bob" so we say that "Bobby" follows "Bob" (or "Bob" precedes "Bobby") in the lexicographical ordering. This is case 2, and the shorter string always precedes the longer one.

6.12.3 `compareTo` method for string comparison

We have seen that we cannot use `==` and `!=` to compare strings for equality and inequality. Instead we use the `equals` method. Similarly we cannot use `<=`, `<`, `>`, or `>=` to compare two strings. Instead there is a `compareTo` instance method in the `String` class. It has the prototype

Boolean expression	**Meaning if true**
`s1.compareTo(s2) <  0`	`s1` precedes `s2`
`s1.compareTo(s2) <= 0`	`s1` precedes or is equal to `s2`
`s1.compareTo(s2) >  0`	`s1` follows `s2`
`s1.compareTo(s2) >= 0`	`s1` follows or is equal to `s2`
`s1.compareTo(s2) == 0`	`s1` is equal to `s2` (same as `s1.equals(s2)`)
`s1.compareTo(s2) != 0`	`s1` is not equal to `s2` (same as `!s1.equals(s2)`)

Table 6.5: Lexicographical string comparison using `compareTo`

```
public int compareTo(String s)
```

The return value indicates the result of the comparison. A negative value for `s1.compareTo(s2)` means that `s1` precedes `s2`, a zero value means that `s1` and `s2` are equal, and a positive value means that `s1` follows `s2`. The boolean expressions that are used to compare strings are shown in Table 6.5. Here is a simple string comparison class you can try in BlueJ.

Class `StringComparer`

book-projects/chapter6/strings

```
package chapter6.strings; // remove this line if you're not using packages
/**
 * A simple class to illustrate how to compare two strings
 */
public class StringComparer
{
   public String compare(String s1, String s2)
   {
      int result = s1.compareTo(s2);

      if (result < 0)
         return s1 + " precedes " + s2;
      else if (result == 0)
         return s1 + " equals " + s2;
      else
         return s1 + " follows " + s2;
   }
}
```

You can also try the following example using the BeanShell workspace editor.

■ EXAMPLE 6.28 **(String comparison using compareTo method)** Using the BeanShell editor type in the `compare` method from the `StringComparer` class and evaluate it in the workspace. Now try the statements

```
bsh % show();
<true>
```

```
bsh % compare("one", "two");
<one precedes two>
bsh % compare("two", "one");
<two follows one>
bsh %
```

to experiment with the `compareTo` method. ■

6.12.4 Case insensitive string comparison

Sometimes it is useful to do a case-insensitive string comparison. This means that lower case letters are considered equivalent to their upper case counterparts. One way to do this is to convert both strings to lower case and then use `compareTo` as in the following expression

```
s1.toLowerCase()(s2.toLowerCase());
```

However, this is not necessary since there is a `compareToIgnoreCase` method in the `String` class with prototype

```
public int compareToIgnoreCase(String s)
```

that does this case-insensitive comparison.

6.13 Boolean valued methods

Methods that return `boolean` values are called **boolean valued methods**. The `equals` method in the `String` class is a good example. There are other such methods in the `String` class.

■ EXAMPLE 6.29 **(Boolean valued String methods)** The `equalsIgnoreCase` method is like the `equals` method but it ignores the case of the letters. It has the prototype

```
public boolean equalsIgnoreCase(String s)
```

Other examples are the `startsWith` and `endsWith` instance methods that have the prototypes

```
public boolean startsWith(String prefix)
public boolean endsWith(String suffix)
```

The first returns true if the string receiving the message starts with the given `prefix`, and the second returns true if the string receiving the message ends with the given `suffix`. ■

We can also write our own boolean valued methods as the following examples show.

■ EXAMPLE 6.30 **(An isLeapYear method)** For the leap year Example 6.23 we can write a boolean valued method called `isLeapYear` that takes the year as a formal argument and returns `true` or `false`, depending on whether the year is a leap year. The method prototype is

```
public boolean isLeapYear(int y)
```

where the formal argument `y` stands for the year. The method declaration is

```
public boolean isLeapYear(int y)
{
   return (y % 4 == 0) && (y % 100 != 0) || (y % 400 == 0);
}
```

This method could easily be tested in BeanShell as follows

```
bsh % show();
<true>
bsh % boolean isLeapYear(int y)
{ return (y % 4 == 0) && (y % 100 != 0) || (y % 400 == 0); }
bsh % isLeapYear(2000);
<true>
bsh % isLeapYear(2003);
<false>
bsh % isLeapYear(2004);
<true>
bsh %
```

assuming that `show()` is on. ■

■ EXAMPLE 6.31 **(A realRoots method)** Given the quadratic equation $ax^2 + bx + c = 0$, the method

```
public boolean realRoots(double a, double b, double c)
{
   return b*b - 4.0*a*c >= 0;
}
```

returns `true` if the equation has real roots and `false` otherwise. ■

6.14 Error checking techniques

A simple version of the bank account class was developed in Chapter 4, page 106. In that version there were no checks for illegal values of the data fields. For example, the account number must be positive, and the balance must not be negative. Also, it must not be possible to withdraw more than the current balance, or deposit a negative amount.

We can use an if-statement to test for these illegal conditions. In general there are several ways to do the error handling. For the class designer the easiest way is to do no error checking at all! This places the entire burden of error checking on the user of the class. For example, to ensure that a withdrawal is legal, the user would have to write statements such as

```
BankAccount myAccount = new BankAccount(123, "Peter Pascoe", 4050.00);
...
if (myAccount.getBalance() >= 5000)
```

```
{
   myAccount.withdraw(5000);
}
```

On the other hand the programmer could simply decide to use

```
myAccount.withdraw(5000);
```

without checking, and put the account in an illegal state with a negative balance.

It is never a good idea to leave error checking to the user of the class. It is the responsibility of the class designer to ensure that a bank account object can never be in an inconsistent state (bad account number or negative balance). This principle is called **data encapsulation** and it ensures data integrity. The ability to encapsulate data within an object and protect it is one of the most important benefits of object-oriented programming.

Ensuring data integrity in a class can be accomplished by following some simple rules in designing classes:

1. Make all data fields private.
2. Check the validity of all arguments in a constructor and report an error if any of them are illegal.
3. Check the validity of method arguments, especially for methods that can change the values of one or more data fields, and report an error if any of them are illegal.
4. Do not return references to private data fields. This is the unwanted side-effect problem illustrated in Chapter 4 using the `MPoint` and `MCircle` classes.

6.14.1 Reporting errors

There are also several ways to report errors:

1. If an error occurs inside a constructor, such as an illegal argument,
   (a) display an error message and exit the program,
   (b) have the constructor "throw an exception" (to be discussed later)
2. If an error occurs inside a method, such as an illegal argument, leave the data fields unchanged and either
   (a) display an error message and exit the program,
   (b) have the method "throw an exception" (to be discussed later),
   (c) return a boolean value or other error indicator that can be checked by the user,
   (d) do nothing and exit the method.

6.14.2 Using boolean return values and exit to report errors

We can use the `BankAccount` class to illustrate error reporting. For example, the original `withdraw` method is

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

which does no error handling. It could be replaced by the boolean valued method

```
public boolean withdraw(double amount)
{
   boolean amountValid = (0 <= amount) && (amount <= balance);
   if (amountValid)
   {
      balance = balance - amount;
   }
   return amountValid;
}
```

which returns `true` if the amount is valid, and `false` otherwise. This method changes the balance only if there is enough money. This ensures data integrity, but leaves the processing of the error to the user. For example, the user could write statements such as

```
BankAccount myAccount = new BankAccount(123, "Andy Dalziel", 4050.00);
...
boolean ok = myAccount.withdraw(5000);
if (! ok)
{
   // report error here
   // ask for a new withdrawal amount or cancel withdrawal
}
```

The important idea here is that the program can recover from the error by either asking for a new withdrawal amount or canceling the withdrawal. Even if the user forgets to do the error checking, by ignoring the return value as in

```
myAccount.withdraw(5000);
```

the data integrity of the object is maintained. Similarly, the original `deposit` method

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

should not change the balance if a negative amount is used, so it could be replaced by

```
public boolean deposit(double amount)
{
   boolean amountValid = amount >= 0;
   if (amountValid)
   {
      balance = balance + amount;
   }
   return amountValid;
}
```

which does not change the balance, in case `amount` is negative.

6.15 Error reporting using exceptions

We now show how to **throw an exception** to indicate illegal arguments in a constructor or method. This is the most important error-handling technique.

6.15.1 Exception classes and objects

An **exception** is an object from an exception class that has error information in it. To throw one means to suspend execution of the program and either process the error or signal the caller of the constructor or method that an error condition has occurred. The `throw` statement is used to throw exceptions. It has the syntax

`throw` *exceptionObject* `;`

where *exceptionObject* is an exception object constructed from one of the exception classes. There are many kinds of exception classes in Java and we can even write our own.

6.15.2 Throwing exceptions in the `BankAccount` class

We now illustrate exceptions for illegal method and constructor arguments using the `BankAccount` class. The exception class is called `IllegalArgumentException` and it already exists in package `java.lang`.

In our first attempt at the `BankAccount` class (page 106) the constructor declaration is

```
public BankAccount(int accountNumber, String ownerName, double initialBalance)
{
   number = accountNumber;
   name = ownerName;
   balance = initialBalance;
}
```

We need to check for account numbers that are 0 or negative, names that have no object associated with them or are empty strings, and negative initial balances. This can be done by throwing an exception if one of these errors occur.

Here is a version of the bank account constructor that constructs exception objects and throws them. An exception constructor can take one argument, a string that specifies an appropriate error message.

```
public BankAccount(int accountNumber, String ownerName, double initialBalance)
{
   if (accountNumber <= 0)
      throw new IllegalArgumentException("Account number must be positive");
   if (ownerName.equals("") || ownerName == null)
      throw new IllegalArgumentException("Owner name not defined");
   if (initialBalance <= 0)
      throw new IllegalArgumentException("Balance must be non-negative");

   number = accountNumber;
   name = ownerName;
   balance = initialBalance;
}
```

Each if-statement checks one of the three arguments. We don't need to use `else` here: when an exception is thrown control is immediately transferred out of the constructor to the method that called it. Since `ownerName` is a reference to a `String` object, we can check if an object has been defined, by comparing the reference with the special value `null`. If `ownerName` is `null` this means that the caller has forgotten to initialize the reference for the string.

We can also use this technique to write the following versions of the `withdraw` and `deposit` methods:

```
public void deposit(double amount)
{
   if (amount < 0)
      throw new IllegalArgumentException("Invalid amount for deposit");
   balance = balance + amount;
}

public void withdraw(double amount)
{
   if (amount < 0 || amount > balance)
      throw new IllegalArgumentException("Invalid amount for withdraw");
   balance = balance - amount;
}
```

When an exception is thrown the Java interpreter immediately stops executing the method or constructor in which the exception occurred and looks for an error processing block called a **catch block**. If it doesn't find one it looks for a catch block in the caller of this method or constructor and so on. Eventually, if the exception is not caught, the Java interpreter will catch it, display the error message, and terminate the program.

In any case when throwing exceptions as in the `BankAccount` example the important idea is that we do not need to concern ourselves with who catches the exception or how it is processed.

We simply provide an informative error message. Here is the complete version of the bank account class with error checking using exceptions.

Class `BankAccount`

book-projects/chapter6/bank_account

```java
package chapter6.bank_account; // remove this line if you’re not using packages
/**
 * A bank account object encapsulates the account number, owner name, and
 * current balance of a bank account.
 * This version checks for illegal method and constructor arguments.
 */
public class BankAccount
{
   private int number;
   private String name;
   private double balance;

   /**
    * Construct a bank account with given account number,
    * owner name and initial balance.
    * @param accountNumber the account number
    * @param ownerName the account owner name
    * @param initialBalance the initial account balance
    * @throws IllegalArgumentException if account number is negative,
    * owner name is null or empty, or if balance is negative.
    */
   public BankAccount(int accountNumber, String ownerName, double initialBalance)
   {
      if (accountNumber <= 0)
         throw new IllegalArgumentException("Account number must be positive");
      if (ownerName.equals("") || ownerName == null)
         throw new IllegalArgumentException("Owner name not defined");
      if (initialBalance < 0)
         throw new IllegalArgumentException("Balance must be non-negative");
      number = accountNumber;
      name = ownerName;
      balance = initialBalance;
   }

   /**
    * Deposit money in the account.
    * @param amount the deposit amount. If amount <= 0 the
    * account balance is unchanged.
    * @throws IllegalArgumentException if deposit amount is negative
    */
   public void deposit(double amount)
   {
      if (amount < 0)
         throw new IllegalArgumentException("Invalid amount for deposit");
      balance = balance + amount;
   }

   /**
    * Withdraw money from the account.
    * If account would be overdrawn the account balance is unchanged.
    * @param amount the amount to withdraw.
    * @throws IllegalArgumentException if withdraw amount is invalid
    */
   public void withdraw(double amount)
   {
      if (amount < 0 || amount > balance)
         throw new IllegalArgumentException("Invalid amount for withdraw");
      balance = balance - amount;
   }

   /**
    * Return the account number.
    * @return the account number.
    */
   public int getNumber()
   {
      return number;
   }

   /**
    * Return the owner name.
    * @return the owner name.
    */
   public String getName()
   {
      return name;
   }

   /**
    * Return the account balance.
    * @return the account balance.
    */
   public double getBalance()
   {
      return balance;
   }

   /**
    * string representation of this account.
    * @return string representation of this account.
    */
   public String toString()
   {
       return "BankAccount[" + number + ", " + name + ", " + balance + "]";
   }
}
```

To test the error processing in this version of `BankAccount` we can use a simple tester class such as

Class `ExceptionTester`

book-projects/chapter6/bank_account

```
package chapter6.bank_account; // remove this line if you're not using packages
/**
 * Showing uncaught exception messages;
 */
public class ExceptionTester
{
   public void doTest()
   {
      BankAccount b = new BankAccount(123, "Fred", 100);
      b.withdraw(200);
   }

   public static void main(String[] args)
   {
      new ExceptionTester().doTest();
   }
}
```

If you execute this class in `BlueJ` the appropriate `throw` statement will be highlighted in the `BankAccount` source code. The following output shows what happens when the Java interpreter processes an illegal argument exception.

```
Exception in thread "main" java.lang.IllegalArgumentException:
Invalid amount for withdraw
at chapter6.bank_account.BankAccount.withdraw(BankAccount.java:58)
at chapter6.bank_account.ExceptionTester.doTest(ExceptionTester.java:10)
at chapter6.bank_account.ExceptionTester.main(ExceptionTester.java:15)
```

Our custom error message is displayed and there is useful information concerning the location of the error. The error occurred on line 58 in the `BankAccount` class within the body of the `withdraw` method which was called from line 10 in the `ExceptionTester` class, and this is within the body of the `doTest` method. Finally, the `doTest` method was called from line 15 which is within the body of the `main` method. Thus, the exception processing traces the flow of execution until the exception occurs.

6.15.3 Catching exceptions

Having the Java interpreter catch exceptions and terminate our program is rarely satisfactory, unless we cannot recover from the error. To catch and process exceptions ourselves we use what is called a **try-catch statement**. The simplest form is given by

```
try
```

```
      {
         // statements to try
      }
      catch ( ... )
      {
         // statements to execute when something in try fails
      }
      // other statements
```

The idea here is that we put any statements that can throw exceptions in the try block. If no exception is thrown the catch block is ignored and control passes to the other statements below the try-catch statement as though the try-catch block was not there. However, if an exception is thrown execution immediately leaves the try block and the statements in the catch block are executed.

For our `BankAccount` example the catch block will have the form

```
      catch (IllegalArgumentException e)
      {
         // statements to execute when try fails
      }
```

Exceptions, like almost everything in Java, are objects. This catch block specifies the class to which the exception belongs, and a name for the exception object. Multiple catch blocks can be used if there is more than one type of exception. We can find out more about the exception that occurred by invoking the `getMessage` method on `e`. A string containing the error message will be returned.

As a simple illustration the following class uses a try-catch block to display the error message for the exception thrown by the `withdraw` method.

Class `ExceptionCatcher`

book-projects/chapter6/bank_account

```
package chapter6.bank_account; // remove this line if you’re not using packages
/**
 * Catching an exception
 */
public class ExceptionCatcher
{
   public void doTest()
   {
      try
      {
         BankAccount b = new BankAccount(123, "Fred", 100);
         b.withdraw(200);
      }
      catch (IllegalArgumentException e)
      {
         System.out.println(e.getMessage());
      }
   }

   public static void main(String[] args)
   {
      new ExceptionCatcher().doTest();
   }
}
```

Here the method call expression `b.withdraw(200)` will throw an exception.

In this simple example we just display the error message using `e.getMessage()`: When the `doTest` method is executed within BlueJ our error message

```
      Invalid amount for withdraw
```

is displayed in the terminal window. If you also want to trace the location of the exception then replace the `println` statement with the statement

```
      e.printStackTrace();
```

In more realistic programs we could try to recover from the error and ask for another withdrawal amount.

6.16 Paper, scissors, rock game (PSR)

As an example of if-statements, boolean-valued methods, and exception processing we consider the paper, scissors, rock (PSR) game. This is a game for two players, called Player 1 and Player 2. We can imagine that each player has a piece of paper (P), a pair of scissors (S), and a rock (R). At the signal, each player will present one of these three items. The rules are simple.

6.16.1 Rules of the game

- If Player 1 chooses paper and Player 2 chooses rock, Player 1 wins (paper covers rock)
- If Player 1 chooses paper and Player 2 chooses scissors, Player 2 wins (scissors cut paper)
- If Player 1 chooses rock and Player 2 chooses scissors, Player 1 wins (rock breaks scissors)
- If both players choose the same item, then the game is a draw.

The nine possible combinations are shown in Table 6.6. The three rows correspond to Player 1's choices, and the columns correspond to Player 2's choices.

6.16.2 Object-oriented PSR game

We can write an OOP version of this game using two classes: one called `PSRPlayer` that represents a player, and one called `PSRGame` that represents the game.

		Player 2		
		Paper	Scissors	Rock
Player 1	Paper	Draw	Player 2	Player 1
	Scissors	Player 1	Draw	Player 2
	Rock	Player 2	Player 1	Draw

Table 6.6: Table of outcomes for the paper, scissors, rock game

Designing the `PSRPlayer` class

We can represent the player choices using the characters `'P'` for paper, `'S'` for scissors, and `'R'` for rock. Each player object represents the current choice so we can provide methods for getting and setting the choice. This gives the class design

```
public class PSRPlayer
{
   private char choice;
   public PSRPlayer() {...}
   public char getChoice() {...}
   public void setChoice(char choice) {...}
}
```

Designing the `PSRGame` class

The `PSRGame` class needs to construct a game for two players and their choices (aggregation), play a single round and report who won. Internally the class will determine the winner based on the rules of the game. The class design is

```
public class PSRGame
{
   private PSRPlayer p1, p2;

   public static final int DRAW = 0;
   public static final int WIN_PLAYER_ONE = 1;
   public static final int WIN_PLAYER_TWO = 2;

   public PSRGame(PSRPlayer p1, PSRPlayer p2) {...}
   public int playRound() {...}
}
```

Here we have used the `int` return type on `playRound` since there are three possible outcomes of a round: player 1 wins, player 2 wins, or the round is a draw. We have defined three constants in the class to represent these three outcomes, and `playRound` will return one of them.

`PSRPlayer` implementation

The `PSRPlayer` implementation is simple:

Class PSRPlayer

book-projects/chapter6/psr_game

```
package chapter6.psr_game; // remove this line if you’re not using packages
/**
 * This class represents a player in the PSR game
 */
public class PSRPlayer
{
   private char choice; // the player’s choice

   /**
    * Construct a PSR game player
    */
   public PSRPlayer()
   {
   }

   /**
    * get the player’s choice
    * @return the player’s choice character
    */
   public char getChoice()
   {
       return choice;
   }

   /**
    * Set the choice made by the player
    * @param choice the choice character
    */
   public void setChoice(char choice)
   {
       this.choice = Character.toUpperCase(choice);
   }
}
```

Here we have converted the player’s character to upper case to make the character choices case-insensitive.

PSRGame implementation

The constructor implementation is simple and the `playRound` method can be expressed in terms of a boolean valued `isWin` method as

```
      public int playRound()
      {
         // throw exceptions here for invalid choices

         if (isWin(p1, p2))
         {
            return WIN_PLAYER_ONE;
         }
         else if (isWin(p2, p1))
         {
            return WIN_PLAYER_TWO;
         }
         else
         {
            return DRAW;
         }
      }
```

Here the `isWin` method returns true if the first player wins over the second player so it is called twice to determine if a player wins. It can be implemented as a static method that uses Table 6.6 to determine who wins. The `playRound` method is also a good place to check for illegal input characters for the choices. We can do this by throwing an exception. The appropriate exception in this case is called `IllegalStateException`.

Finally, Table 6.6 shows that there are three ways that Player 1 can win, so we can simply use a “logical or” to combine them to obtain the condition.

```
      (pc1 == ’P’ && pc2 == ’R’)      // row 1
      || (pc1 == ’S’ && pc2 == ’P’)   // row 2
      || (pc1 == ’R’ && pc2 == ’S’)   // row 3
```

where `pc1` and `pc2` represent the choices of Player 1 and Player 2.

Here is the complete implementation.

Class `PSRGame`

book-projects/chapter6/psr_game

```
package chapter6.psr_game; // remove this line if you’re not using packages
/**
 * Play one round of the PSR game with two players.
 */
public class PSRGame
{
   public static final int WIN_PLAYER_ONE = 1;
   public static final int WIN_PLAYER_TWO = 2;
   public static final int DRAW = 0;

   private PSRPlayer p1, p2;

   /**
    * Construct a game from two players
    * @param p1 the first player
    * @param p2 the second player
    */
   public PSRGame(PSRPlayer p1, PSRPlayer p2)
   {
       this.p1 = p1;
       this.p2 = p2;
   }

   /**
    * Return the result of one round of the game as one of
    * the three public constants.
    * @return outcome as one of the three public constants
    */
   public int playRound()
   {
       if ( isInvalidChoice(p1) )
       {
           throw new IllegalStateException("Player 1: invalid choice");
       }

       if ( isInvalidChoice(p2) )
       {
           throw new IllegalStateException("Player 2: invalid choice");
       }

       if (isWin(p1, p2))
       {
           return WIN_PLAYER_ONE;
       }
       else if (isWin(p2, p1))
       {
           return WIN_PLAYER_TWO;
       }
       else
       {
           return DRAW;
       }
   }

   /* Return true if the first player p1 wins over the second player p2.
    */
   private static boolean isWin(PSRPlayer p1, PSRPlayer p2)
   {
       char pc1 = p1.getChoice();
       char pc2 = p2.getChoice();
       return  (pc1 == 'P' && pc2 == 'R') || (pc1 == 'S' && pc2 == 'P') ||
           (pc1 == 'R' && pc2 == 'S');
   }


   private static boolean isInvalidChoice(PSRPlayer p)
   {
       char choice = p.getChoice();
       return choice != 'P' && choice != 'S' && choice != 'R';
   }
}
```

Testing the class with BlueJ

You can play the game in BlueJ as follows:

1. Construct two player objects called `p1` and `p2`.
2. Construct a `PSRGame` object called `game`, using `p1` and `p2` as arguments.
3. Select the `setChoice` method from the `PSRPlayer` object menu for each player and make a choice.
4. Select the `playRound` choice from the `PSRGame` object menu. The result 0, 1, or 2 will be shown in a method result box.
5. Repeat steps 3 and 4 for another round.

Testing the class with BeanShell

The following example shows how the class can be tested using BeanShell.

■ EXAMPLE 6.32 **(PSR game)** Try the statements in BeanShell

```
bsh % addClassPath("c:/book-projects/chapter6/psr_game");
bsh % PSRPlayer p1 = new PSRPlayer();
bsh % PSRPlayer p2 = new PSRPlayer();
bsh % PSRGame game = new PSRGame(p1, p2);
bsh % p1.setChoice('p');
bsh % p2.setChoice('s');
bsh % print(game.playRound());
2
bsh % p1.setChoice('s');
bsh % p2.setChoice('p');
bsh % print(game.playRound());
1
bsh % p1.setChoice('p');
bsh % p2.setChoice('p');
bsh % print(game.playRound());
0
bsh %
```

to play three rounds of the game. ■

Running the game using a `main` method

To play a round of this game using a `main` method in BlueJ, or outside BlueJ from the command line, requires that we learn how to get console (terminal) input from the user. We need to write a special kind of **user interface** class called a **console interface** to play one round of the game from the command line as follows:

```
Player 1, enter your choice: P, S, or R
p
Player 2, enter your choice: P, S, or R
s
Player 2 wins!
```

Here each player types a character and presses return in response to a prompt.

6.17 Console Input Using a `Scanner` object

Before Java 1.5 getting console input was not trivial. Most people wrote a special class to do this. Console input has been standardized in Java 1.5 with the introduction of the `Scanner` class. It is in a package called `java.util` and can be imported into any class by using the `import` statement

```
import java.util.Scanner;
```

6.17.1 Some useful `Scanner` methods

This class has a constructor and methods with the following prototypes

- **`Scanner input = new Scanner(System.in);`**

  Construct a `Scanner` object called `input` and connect it for reading from the console. There are many other types of constructors but this is the only one we need. The `System.in` object is used to get input from the keyboard in the console window just as `System.out` is used to display output in the console window.

- **`public int nextInt()`**

  Read the next number typed in the console window and return it as an `int` value.

- **`public long nextLong()`**

  Read the next number typed in the console window and return it as a `long` value.

- **`public float nextFloat()`**

  Read the next number typed in the console window and return it as a `float` value.

- **`public double nextDouble()`**

  Read the next number typed in the console window and return it as a `double` value.

- **`public String nextLine()`**

  Read the rest of a line typed in the console window and return it as a `String`.

There are many other methods but these are the only ones we need.

6.17.2 One input per line input model

If you are not careful there are some pitfalls when using the `Scanner` class for interactive input. For example, the statements

```
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
System.put.println("Enter your name");
String name = input.nextLine();
```

do not work as you may expect since `name` will be the empty string. This is so because when you enter the age and press the Enter key the newline is not read by the `nextInt` method. Instead it is read by the `nextLine` method and the result is the empty string being assigned to `name`. The name that you typed is left unread in the input buffer.

The following statements avoid this

```
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
input.nextLine(); // eat the end of line
System.put.println("Enter your name");
String name = input.nextLine();
```

Following each numeric input method call by the statement

```
input.nextLine();
```

will eat the new line character (throw it away). This is called the one input per line interactive input model.

6.17.3 Console interface class for the PSR game

Using the `Scanner` class we can write a console-interface class for the PSR game which can be run both inside `BlueJ` and outside `BlueJ` from the command line.

Class `PSRGameRunner`

book-projects/chapter6/psr_game

```
package chapter6.psr_game; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A console interface for one round of the PSRGame
 */
public class PSRGameRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      PSRPlayer p1 = new PSRPlayer();
      PSRPlayer p2 = new PSRPlayer();
      PSRGame game = new PSRGame(p1, p2);

      System.out.println("Player 1, enter your choice: P, S, or R");
      char player1Choice = input.nextLine().charAt(0);
      p1.setChoice(player1Choice);

      System.out.println("Player 2, enter your choice: P, S, or R");
      char player2Choice = input.nextLine().charAt(0);
      p2.setChoice(player2Choice);

      int result = 0; // any initial value will do

      try
      {
         result = game.playRound();
      }
      catch (IllegalStateException e)
      {
         System.out.println("Illegal input");
         result = PSRGame.DRAW;
      }

      if (result == PSRGame.WIN_PLAYER_ONE)
         System.out.println("Player 1 wins!");
      else if (result == PSRGame.WIN_PLAYER_TWO)
         System.out.println("Player 2 wins!");
      else
         System.out.println("It’s a draw");
   }

   public static void main(String[] args)
   {
      PSRGameRunner program = new PSRGameRunner();
      program.run();
   }
}
```

In the `run` method we first construct the objects needed to play a round. Then we use the `nextLine` method and the `setChoice` method to set each player's character to the first character of the input line.

If `playRound` throws an exception, we display a message and set the outcome to a draw. It is necessary to declare the `result` variable outside the `try` block, otherwise it would be local to the try block and undefined in the if-statement that displays the result.

This `PSRGameRunner` class can be run from the command line or inside `BlueJ` using either the `main` method or by constructing a `PSRGameRunner` object and choosing its `run` method.

6.18 Complex roots of a quadratic equation

We have written a `QuadraticRootFinder` class (page 256) that calculates the roots only if they are real. We now generalize and find the roots in any case, real or complex.

Recall that a complex number has the form $a+bi$ where a and b are called the real and imaginary part of the complex number and $i = \sqrt{-1}$, $i^2 = -1$. Given the quadratic equation $ax^2+bx+c=0$ the roots have non-zero imaginary part if $b^2-4ac<0$ and are given by

$$r_1 = c+di, \quad r_2 = c-di \quad \text{where } c = -\frac{b}{2a}, \qquad d = \frac{\sqrt{|b^2-4ac|}}{2a}$$

An interesting way to find the roots is to first write a class called `Complex` that represents complex numbers and then use it in a class called `ComplexQuadraticRootFinder` to find the roots.

A simple design for the `Complex` class is

```
public class Complex
{
   private double realPart;
   private double imagPart;

   public Complex(double real, double imag) {...}
   public double getRealPart() {...}
   public double getImagPart() {...}
   public String toString() {...}
```

6.18.1 `Complex` class

Except for the `toString` method the implementation is straightforward:

Class `Complex`

book-projects/chapter6/root_finder

```
package chapter6.root_finder; // remove this line if you’re not using packages
/**
 * A simple class whose objects represent complex numbers by their
 * real and imaginary parts.
 */
public class Complex
{
   private double realPart;
   private double imagPart;

   /** Construct complex number with given real and imaginary parts.
    * @param real the real part of the complex number
    * @param imag the imag part of the complex number
    */
   public Complex(double real, double imag)
   {
      realPart = real;
      imagPart = imag;
   }

   /**
    * Returns real part of this complex number.
    * @return the real part of this complex number
    */
   public double getRealPart()
   {
      return realPart;
   }

   /**
    * Returns imaginary part of this complex number.
    * @return the imaginary part of this complex number
    */
   public double getImagPart()
   {
      return imagPart;
   }

   /** Returns a string representation of this complex number.
    * @return a string representation of this complex number
    * of the form a + b i, a - b i, or a in case b = 0
    */
   public String toString()
   {
      if (imagPart > 0)
         return realPart + " + " + imagPart + " i";
      else if (imagPart < 0)
         return realPart + " - " + Math.abs(imagPart) + " i";
      else
         return realPart + "";
   }
}
```

The `toString` method displays the complex number in the form $a+bi$ or $a-bi$ depending on the sign of b.

Now the `QuadraticRootFinder` class (page 6.4.1) can easily be modified to give the complex version:

Class `ComplexQuadraticRootFinder`

book-projects/chapter6/root_finder

```
package chapter6.root_finder; // remove this line if you're not using packages
/**
 * An object of this class can calculate the complex roots of the
 * quadratic equation ax^2 + bx + c = 0 given the coefficients a, b, and c.
 */
```

```
public class ComplexQuadraticRootFinder
{
   private double a, b, c;
   private Complex root1, root2;

   /**
    * Construct a quadratic equation root finder given the coefficients
    * @param aCoeff first coefficient in ax^2 + bx + c
    * @param bCoeff second coefficient in ax^2 + bx + c
    * @param cCoeff third coefficient of ax^2 + bx + c
    */
   public ComplexQuadraticRootFinder(double aCoeff, double bCoeff, double cCoeff)
   {
      a = aCoeff;
      b = bCoeff;
      c = cCoeff;
      doCalculations();
   }

   private void doCalculations()
   {
      double d1 = b*b - 4*a*c;
      double d2 = Math.sqrt(Math.abs(d1));

      if (d1 >= 0)
      {
         // real root case

         double realPart1 = (-b - d2) / (2.0 * a);
         double realPart2 = (-b + d2) / (2.0 * a);
         root1 = new Complex(realPart1, 0.0);
         root2 = new Complex(realPart2, 0.0);
      }
      else
      {
         // complex root case

         double realPart = -b / (2.0 * a);
         double imagPart = d2 / (2.0 * a);
         root1 = new Complex(realPart, imagPart);
         root2 = new Complex(realPart, -imagPart);
      }
   }

   /**
    * Return the first root as a complex number.
    * @return the first root as a complex number
    */
   public Complex getRoot1()
   {
       return root1;
   }
```

```
/**
 * Return the second root as a complex number.
 * @return the second root as a complex number
 */
public Complex getRoot2()
{
    return root2;
}

/**
 * Return the coefficient of xˆ2.
 * @return the coefficient of xˆ2
 */
public double getA()
{
   return a;
}

/**
 * Return the coefficient of x.
 * @return the coefficient of x
 */
public double getB()
{
   return b;
}
/**
 * Return the constant coefficient.
 * @return the constant coefficient
 */
public double getC()
{
   return c;
}

/**
 * Change the value of the coefficient of xˆ2.
 * @param value the new value for the coefficient of xˆ2
 */
public void setA(double value)
{
   a = value;
   doCalculations();
}

/**
 * Change the value of the coefficient of x.
 * @param value the new value for the coefficient of x
 */
public void setB(double value)
{
      b = value;
      doCalculations();
   }

   /**
    * Change the value of the constant coefficient.
    * @param value the new value for the constant coefficient.
    */
   public void setC(double value)
   {
      c = value;
      doCalculations();
   }
}
```

The return type of the `getRoot1` and `getRoot2` methods is now `Complex`.

Testing the class with BlueJ

To test the class in BlueJ perform the following steps.

1. Construct a `ComplexQuadraticRootFinder` object called `finder` with values for *a*, *b*, *c*.
2. From the object menu select `getRoot1()` and click on `<object-reference>`.
3. Click "Get" button and name the `Complex` object `root1`. It will appear on the object bench.
4. From the object menu select `getRoot2()` and click on `<object-reference>`.
5. Click "Get" button and name the `Complex` object `root2`. It will appear on the object bench.
6. From the object menu of `root1` and `root2` select the `toString` method to see the roots.

Testing the class with BeanShell

The following example shows how the class can be tested using BeanShell.

■ EXAMPLE 6.33 **(Complex roots)** Try the statements in BeanShell

```
bsh % addClassPath("c:/book-projects/chapter6/root_finder");
bsh % ComplexQuadraticRootFinder finder = new
ComplexQuadraticRootFinder(3,4,5);
bsh % Complex r1 = finder.getRoot1();
bsh % Complex r2 = finder.getRoot2();
bsh % print(r1);
-0.6666666666666666 + 1.1055415967851332 i
bsh % print(r2);
-0.6666666666666666 - 1.1055415967851332 i
bsh %
```

to find the complex roots of a quadratic equation. ■

Console interface

We can use `Scanner` to write the following console interface called `ComplexRunner` for the complex root finding class:

Class `ComplexRunner`

book-projects/chapter6/root_finder

```
package chapter6.root_finder; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A simple runner class for finding complex roots of a quadratic equation
 */
public class ComplexRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter coefficient a");
      double a = input.nextDouble();
      input.nextLine(); // eat the end of line

      System.out.println("Enter coefficient b");
      double b = input.nextDouble();
      input.nextLine(); // eat the end of line

      System.out.println("Enter coefficient c");
      double c = input.nextDouble();
      input.nextLine(); // eat the end of line

      ComplexQuadraticRootFinder finder = new ComplexQuadraticRootFinder(a,b,c);

      Complex root1 = finder.getRoot1();
      Complex root2 = finder.getRoot2();

      System.out.println("Root 1 is " + root1);
      System.out.println("Root 2 is " + root2);
   }

   public static void main(String[] args)
   {
      ComplexRunner runner = new ComplexRunner();
      runner.run();
   }
}
```

This class can be run inside `BlueJ` using the `run` method and the terminal window, or from the command line using the `main` method. Typical command-line output is

```
Enter coefficient a
3
```

```
Enter coefficient b
4
Enter coefficient c
5
Root 1 is -0.6666666666666666 + 1.1055415967851332 i
Root 2 is -0.6666666666666666 - 1.1055415967851332 i
```

6.19 Review exercises

► **Review Exercise 6.1** Define the following terms and give examples of each.

conditional execution	boolean expression	boolean literal
comparison expression	comparison operator	equality expression
equality operator	relational expression	relational operator
block	flowchart	absolute error
relative error	conditional operator	compound boolean expression
truth table	short circuit evaluation	lexicographical ordering
character code	boolean valued method	exception
throwing an exception	catch block	

► **Review Exercise 6.2** Develop a set of test data for program class `PSRGameTester` that will guarantee that (1) every branch of the `PSRGame` program is tested, and (2) for all legal inputs the program produces the correct output. This is called exhaustive testing and it constitutes a proof that the program is correct. In most cases programs are too complicated for exhaustive testing.

► **Review Exercise 6.3** Write truth tables that verify deMorgan's laws

$$\begin{aligned} \sim(a \wedge b) &= (\sim a) \vee (\sim b) \\ \sim(a \vee b) &= (\sim a) \wedge (\sim b) \end{aligned}$$

► **Review Exercise 6.4** The operation $p \vee q$ is true if either of p and q is true or both are true. The **exclusive or** of p and q, denoted by $p \oplus q$, excludes the case that both are true. Its truth table is shown in Table 6.7. Using a truth table show that

p	q	$p \oplus q$
false	false	false
false	true	true
true	false	true
true	true	false

Table 6.7: Truth table for the exclusive or

$$p \oplus q \equiv (p \vee q) \wedge \sim(p \wedge q)$$

where $a \equiv b$ means that a and b are logically equivalent (they have the same truth table). This shows that the exclusive or can be expressed in terms of the logical operators $\wedge$, $\vee$, and $\sim$.

► **Review Exercise 6.5** For the `CircleCalculator` class from Chapter 3 (page 63) what modifications would you make to include error processing using exceptions.

► **Review Exercise 6.6** For the `TriangleCalculator` class from Chapter 3 (page 64) what modifications would you make to include error processing using exceptions.

6.20 BeanShell exercises

The following BeanShell exercises can be done using the Workspace Editor. First run BeanShell, then choose "Workspace Editor" from the "File" menu to open the editor.

Now you can type statements into the editor and they won't be executed as they are entered. When you have finished entering statements choose "Evaluate in Workspace" from the "Evaluate" menu. Now the statements will be executed. You can edit the statements and evaluate them again, and so on.

This is useful for testing static methods. Type in the method, evaluate it then test it interactively using the workspace.

► **BeanShell Exercise 6.1** Evaluate the `abs` method in Example 6.2 in the editor and try statements such as the following in the workspace

```
bsh % double result = abs(3);
bsh % print(result);
3.0
bsh % result = abs(-3);
bsh % print(result);
3.0
```

► **BeanShell Exercise 6.2** Do BeanShell Exercise 6.1 using the `abs` method in Example 6.10.

► **BeanShell Exercise 6.3** Repeat BeanShell Exercise 6.1 for the `cubeRoot` method in Example 6.3.

► **BeanShell Exercise 6.4** Do BeanShell Exercise 6.3 using the `cubeRoot` in Example 6.12.

► **BeanShell Exercise 6.5** Using the statements in Example 6.6, write a method in the editor called `max` that takes two `double` values and returns their maximum. Test the method using the workspace as in BeanShell Exercise 6.1.

► **BeanShell Exercise 6.6** Do BeanShell Exercise 6.5 using the statement in Example 6.11.

► **BeanShell Exercise 6.7** Convert Example 6.13 into a method called `calculateTax` that has one argument, the amount `a`, and returns the tax calculated. Test your method as in BeanShell Exercise 6.1.

► **BeanShell Exercise 6.8** Do BeanShell Exercise 6.7 using the if-statement in in Example 6.22.

► **BeanShell Exercise 6.9** Convert Example 6.14 into a method called `letterGrade` that has one argument for the mark, and returns the letter grade as a `String`. Test your method as in BeanShell Exercise 6.1.

► **BeanShell Exercise 6.10** Do BeanShell Exercise 6.9 using the if-statement in in Example 6.21.

► **BeanShell Exercise 6.11** Verify some of the results in Table 6.4 using BeanShell statements such as

```
bsh % print((int) 'a');
97
bsh % print((int) 'A');
65
```

6.21 Programming exercises

► **Exercise 6.1 (A marks converter class)**
Write a class called `MarksConverter` that uses a method with prototype

```
public String letterGrade(int mark)
```

based on the if-statement in Example 6.21, to convert a mark to a letter grade. Indicate how to test your class in BlueJ and BeanShell.

► **Exercise 6.2 (A tax calculator class)**
Write a class called `TaxCalculator` that uses a method with prototype

```
public double calculateTax(double amount)
```

based on the if-statement in Example 6.22, to calculate the tax on a given amount of money. Indicate how to test your class in BlueJ and BeanShell.

► **Exercise 6.3 (A better `ChangeHelper` class)**
Rewrite the `ChangeHelper` class in Exercise 3.6, Chapter 3, so that zero amounts are not displayed. You should also check that the amount received is not smaller than the amount due.

► **Exercise 6.4 (A better `CircleCalculator` class)**

(a) Write a new version of the `CircleCalculator` class from Chapter 3 (page 63) that uses an exception in case there is an illegal argument in the constructor.

(b) Write a console-interface called `CircleCalculatorRunner` that shows how to test this class using the `Scanner` class.

► **Exercise 6.5 (A better `TriangleCalculator` class)**

(a) Write a new version of the `TriangleCalculator` class from Chapter 3 (page 64) that uses exceptions in case there are illegal arguments

(b) Write a console-interface called `TriangleCalculatorRunner` that shows how to test this class using the `Scanner` class.

► Exercise 6.6 (Maximum of three numbers)
Write a class called `MaxThreeCalculator` that computes the maximum of three `double` numbers using the following class design.

```
public class MaxThreeCalculator
{
   private double maximum; // maximum of x1, x2, and x3

   public MaxThreeCalculator(double x1, double x2, double x3) {...}
   public double getMaximum() { return maximum; }
}
```

Use two private methods with prototypes

```
private double max2(double n1, double n2)
private double max3(double n1, double n2, double n3)
```

It doesn't matter whether these methods are static or not since they don't access any instance data fields. The `max2` method should return the maximum of two numbers and the `max3` method should use `max2` to return the maximum of three numbers. Finally, `max3` can be called in the constructor to calculate the value of the data field `maximum` which can be returned by the `getMaximum` method.

Give a set of test data that you would use to verify the correctness of your program.

► Exercise 6.7 (Finding the smallest of three strings)

(a) Write a class called `StringSorter` that takes three strings and arranges them in increasing lexicographical order using the `compareTo` method in the `String` class. Use the class design

```
public class StringSorter
{
   private String first, second, third;
   public StringSorter(String x1, String x2, String x3) {...}
   public String getFirst() { return first; }
   public String getSecond() { return first; }
   public String getThird() { return first; }
}
```

The constructor should sort the three strings `x1`, `x2`, and `x3` and assign them in sorted order to `first`, `second`, and `third`.

(b) Give a set of test data that you would use to verify the correctness of your program.

(c) Write a console-interface called `StringSorterTester` that shows how to test this class using the `Scanner` class.

► **Exercise 6.8 (Finding the maximum balance for three bank accounts)**
Write a class called `MaxMinAccount` that finds the minimum and maximum balance for three `BankAccount` objects. Use the class design

```
public class MaxMinAccount
{
   private BankAccount min, max;
   public MaxMinAccount(BankAccount b1, BankAccount b2, BankAccount b3) {...}
   public BankAccount getMin() { return min; }
   public BankAccount getMax() { return max; }
}
```

Here `min` is a reference to the account with the minimum balance among `b1`, `b2`, and `b3`, and `max` is a reference to the account with the maximum balance. The "get" methods return references to these accounts.

► **Exercise 6.9 (Multiple if-statement and decision tables)**

(a) Multiple if-statements directly correspond to a decision table and vice versa. Such a table lists a set of mutually exclusive rules for calculating some quantity. As an example, consider the calculation of the sales commission received by a real estate agent for selling a house. Here is the table.

Selling price p	**Commission**
$0 \le p \le \$100,000$	3 percent
$\$100,000 < p \le \$250,000$	5 percent
$\$250,000 < p \le \$500,000$	7 percent
$p > \$500,000$	10 percent

Write a class called `SalesCommissionCalculator` having the structure

```
public class SalesCommissionCalculator
{
   private double sellingPrice;
   private double commission;

   public SalesCommission(double sellingPrice) {...}

   public double getSellingPrice() { return sellingPrice; }
   public double getCommission() { return commission; }
}
```

that has a constructor that uses the selling price to construct a sales commission object and calculate the commission. The two enquiry methods can be used to retrieve the selling price and commission. The constructor should throw an exception if the selling price is negative.

Do the calculation of the commission from the selling price using a method with prototype

```
public double commission(double sellingPrice)
```

that returns the commission. Now the `SalesCommission` constructor can simply call this method.

(b) Use the following console-interface class to test your class.

```
import java.util.Scanner;
public class SalesCommissionRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter selling price");
      double price = input.nextDouble();
      input.nextLine();

      try
      {
         SalesCommission sc = new SalesCommission(price);
         System.out.println("Selling price: " + sc.getSellingPrice());
         System.out.println("Commission: " + sc.getCommission());
      }
      catch (IllegalArgumentException e)
      {
         System.out.println(e.getMessage());
      }
   }

   public static void main(String[] args)
   {
      new SalesCommissionRunner().run();
   }
}
```

► **Exercise 6.10 (Calculating federal income tax)**

(a) Using the previous exercise as a guide, write a `FederalTaxCalculator` class that computes the Federal tax, given two amounts: the taxable income and the total non-refundable tax credits. Here are the mutually exclusive rules:

1. If the taxable income is not greater than $29,590.00, the federal tax is 17 percent of the taxable income.
2. If the taxable income is greater than $29,590.00 but not greater than $59,180.00, the federal tax is $5030.00 on the first $29,590.00 and 26 percent on the remainder.
3. If the taxable income is greater than $59,180.00, the federal tax is $12,724.00 on the first $59,180.00 and 29 percent on the remainder.

From the amount calculated, subtract the total non-refundable tax credits to obtain the total federal tax payable. If this amount is negative, the total federal tax payable is zero.

(b) Write a console-interface called `TaxCalculatorRunner` that reads the taxable income and the total non-refundable tax credits, computes the total federal tax payable, and displays the result.

► **Exercise 6.11 (Calculating roots of quadratic equation)**
Modify the `ComplexQuadraticRootFinder` class to use in the real root case, the following formulas

1. If $b \geq 0$ define $r_1 = -\dfrac{1}{2a}\left(b + \sqrt{b^2 - 4ac}\right)$
2. If $b \leq 0$ define $r_1 = -\dfrac{1}{2a}\left(b - \sqrt{b^2 - 4ac}\right)$
3. In either case $r_1 r_2 = c/a$ so the second root is $r_2 = c/(ar_1)$

Your class should also deal with special cases which arise when some of the coefficients are zero or the roots are not real. For example, if $a = 0$ and $b = 0$ then there is no equation, if $a = 0$ and $b \neq 0$ then the equation is linear and there is one root. The class can provide boolean-valued methods `hasRealRoots`, `isLinear`, `isInvalidEquation` that can be used by a runner class to determine what to display.

Give a set of test cases for the various paths through your class. and write a suitable console-interface class to test your class

► **Exercise 6.12** Roman numerals are still in use. In the motion picture industry the year a film is released is given in roman numerals. Write a program called `RomanNumeralConverter` to convert year numbers in the range 1 to 3999 to a string of roman numerals. Use an instance method with prototype

```
public String roman(int year)
```

For example the year 1998 is MCMXCVIII in roman numerals and the year 2003 is MMIII.

Hint: Use `/` and `%` to determine the thousands, hundreds, tens and units digits of the number and convert each part to roman numerals.

► **Exercise 6.13 (Solving cubic equations)** A mathematician has given you the pseudo-code algorithm for finding the real roots of the cubic equation $ax^3 + bx^2 + cx + d = 0$ shown in Figure 6.8. According to the algorithm the equation has either one real root or three real roots. Write a program class called `CubicSolver` for this algorithm that has the following structure

```
public class CubicSolver
{
   private double a, b, c, d;   // ax^3 + bx^2 + cx + d = 0
   private double root1, root2, root3;   // roots of equation
   private boolean oneRealRoot; // true in one real root case
```

```
**ALGORITHM** CubicSolver($a$, $b$, $c$, $d$)
$p \leftarrow \frac{1}{3a^2}\left[3ac - b^2\right]$,    $q \leftarrow \frac{1}{27a^3}\left[2b^3 - 9abc + 27a^2d\right]$
$\Delta \leftarrow \frac{p^3}{27} + \frac{q^2}{4}$,    $s \leftarrow \frac{b}{3a}$
**IF** $\Delta > 0$ **THEN**
    $f_1 \leftarrow -\frac{q}{2} + \sqrt{\Delta}$,    $f_2 \leftarrow -\frac{q}{2} - \sqrt{\Delta}$
    $y_1 \leftarrow (f_1)^{1/3} + (f_2)^{1/3}$
    $x_1 \leftarrow y_1 - s$
    **RETURN** $x_1$
**ELSE IF** $(\Delta = 0) \wedge (q = 0)$ **THEN**
    $x_1 \leftarrow -s$,    $x_2 \leftarrow -s$,    $x_3 \leftarrow -s$
    **RETURN** $x_1, x_2, x_3$
**ELSE**
    $m \leftarrow 2\sqrt{-\frac{p}{3}}$
    $\theta \leftarrow \frac{1}{3}\arccos\left(\frac{3q}{pm}\right)$
    $y_1 \leftarrow m\cos\theta$,    $y_2 \leftarrow m\cos\left(\theta + \frac{2\pi}{3}\right)$,    $y_3 \leftarrow m\cos\left(\theta + \frac{4\pi}{3}\right)$
    $x_1 \leftarrow y_1 - s$,    $x_2 \leftarrow y_2 - s$,    $x_3 \leftarrow y_3 - s$
    **RETURN** $x_1, x_2, x_3$
**END IF**
```

Figure 6.8: Pseudo-code algorithm for roots of a cubic equation

```
        public CubicSolver(double aa, double bb, double cc, double dd ) {...}
        public boolean hasOneRealRoot() { return oneRealRoot; }
        public boolean isCubic() { return a != 0.0; }
        public double getRoot1() { return root1; }
        public double getRoot2() { return root2; }
        public double getRoot3() { return root3; }
        public double getA() { return a; }
        public double getB() { return b; }
        public double getC() { return c; }
        public double getD() { return d; }

        private double cubeRoot(double x) {...}
    }
```

where the `cubeRoot` method is from Example 6.12. The `QuadraticSolver` class can be used if `isCubic` returns false (coefficient of x^3 is zero).

Write a runner class to test the program and develop some test data. Include a check by also displaying the value of $ax^3 + bx^2 + cx + d$ for each root found (the value should be close to zero).

Chapter 7

Repetition Structures

The while, do-while, and for-statements

Outline

The while-statement (while-loop)

String to integer conversion example

Square root and factorization examples

Sentinel controlled while-loops

Query controlled while-loops

Do-while statement (do-while loop)

General loop structures

For-statement (for-loop)

Computing big factorials

Loan repayment table example

Nested loops

Investment table example

Graphing a function

Recursion and loops

Recursive factorial and gcd examples

```
while ( BooleanExpression )
{
    Statements
}
```

Figure 7.1: A template for the while-statement

7.1 Introduction

In many algorithms we need to repeat the execution of a sequence of one or more statements several times. For a known small number of repetitions you could simply write down the statements to repeat several times. This is not practical if the number of repetitions is large and doesn't work at all if the number of repetitions is not known in advance. Therefore all high-level languages provide one or more **repetition structures** that specify a sequence of statements to be repeated and a condition to continue or terminate the repetitions. A repetition structure is often called a **loop** and is the third and last of the basic programming structures. The first two are the **sequential structure** and the **conditional structure**.

In this chapter we introduce both the pseudo-code and Java forms of the three repetition structures: the while-statement, the do-while statement, and the for-statement. We will see that the for-statement and the do-while statement are just special types of the while-statement that are provided for convenience.

The while and do-while statements are normally used when the number of repetitions is not known in advance but depends on some condition. We discuss several important variations such as the sentinel controlled while-loop, and the query controlled while-loop.

The for-statement is normally used when the number of repetitions can be determined in advance, either as a constant or as the value of an expression.

The connection between loops and simple recursion is also discussed.

7.2 The while-statement (while-loop)

If the number of repetitions is not known in advance but depends on some condition, given by a boolean expression, then the **while-statement** (also called a **while-loop**) can be used. In pseudo-code it can be expressed as

```
WHILE BooleanExpression DO
    Statements
END WHILE
```

In Java the while-loop has the structure shown in Figure 7.1. *BooleanExpression* is a boolean valued expression and *Statements* is a sequence of zero or more statements defining the block to be repeated. As in the case of the if-statement, the parentheses surrounding the boolean expression are necessary and the braces can be omitted only if there is one statement to be repeated.

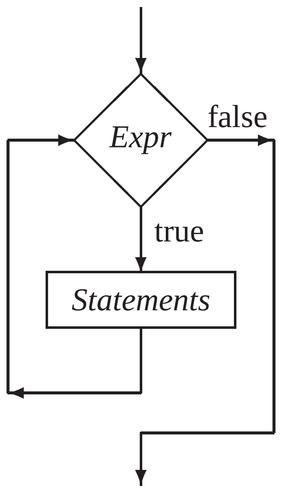


Figure 7.2: A flowchart for the execution of a while-statement

When the while-loop is encountered *BooleanExpression* is evaluated. If the value is true then the statements in the block are executed, otherwise the while-loop is ignored and control resumes below it. Each time the statements in the loop are executed the boolean expression is re-evaluated to determine if the statements should be executed again.

Once the loop is entered the only way out is to have the statements in the body of the loop change the value of one or more of the variables defining the boolean expression, causing it's value to become false. Otherwise we have what is called an **infinite loop**.

A flowchart for the execution of the while-statement is shown in Figure 7.2. It clearly indicates that to avoid an infinite loop there must be a statement in the body of the loop that eventually forces the boolean expression *Expr* to become false.

Here are some simple examples of while-loops.

■ EXAMPLE 7.1 **(Counting up with a while-loop)** The statements

```
int count = 1;
while (count <= 10)
{
   System.out.print(count + " ");
   count = count + 1; // or use count++
}
```

display `1 2 3 4 5 6 7 8 9 10`. Initially, the expression `count <= 10` is true since `count` is initialized to `1`. Therefore the loop is entered and `1` is displayed. Since `count` is incremented each time through the loop, the expression `count <= 10` will become false when it reaches `11`. Therefore the last number displayed is `10`. To try this example using the BeanShell workspace and editor choose "Capture System in/out/err" from the "File" menu. ■

■ EXAMPLE 7.2 **(Counting down with a while-loop)** The statements

```
int count = 10;
while (count >= 1)
{
   System.out.print(count + " ");
   count = count - 1; // or use count--
}
```

count backwards beginning at `10` and display `10 9 8 7 6 5 4 3 2 1`. When `count` is decremented to `0` the boolean expression `count >= 1` becomes false and the loop terminates. Therefore the last number displayed is `1`. To try this example using the `BeanShell` workspace and editor choose “Capture System in/out/err” from the “File” menu. ■

■ EXAMPLE 7.3 **(Loop that may not terminate)** Given the integer $n > 0$ consider the loop

```
long k = n;
System.out.print(k);
while (k > 1)
{
   if (k % 2 == 1) // k is odd
   {
      k = 3*k + 1;
   }
   else // k is even
   {
      k = k / 2;
   }
   System.out.print("," + k);
}
```

Here the value of k is changing each time through the loop but it is not clear that k will eventually become 1 to stop the loop. In fact no one knows for arbitrary $n > 0$ if the loop will terminate. You might become famous if you can prove this for any integer n. If $n = 8$ the loop displays `8,4,2,1` and stops, if $n = 7$ the loop displays `7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1` and stops, and if $n = 27$, then 112 numbers are displayed ending in 1. This loop also ignores the fact that the calculation of `3*k + 1` may produce overflow. This can detected using the if-statement

```
if (k > 3074457345618258602L)
   System.out.println("Overflow has occurred");
```

just before the calculation. The strange long integer literal here is the largest one such that `3*k + 1` does not produce overflow. ■

■ EXAMPLE 7.4 **(Drawing some horizontal lines)** Suppose we want to draw 10 horizontal lines each separated by 20 pixels. Each line should begin at $x = 10$ and end at $x = 200$. The top line should have $y = 10$. Then the left end of line k has coordinates $(10, 10 + 20k)$ for $k = 0, 1, \ldots 9$ and the right end has coordinates $(200, 10 + 20k)$. Assuming that `g2D` is the graphics context (see Chapter 5), the while-loop

```
int k = 0;
while (k <= 9)
{
   double y = 10.0 + 20.0*k; // y coordinate of line k
   g2D.draw(new Line2D.Double(10,y,200,y));
   k = k + 1; // or k++;
}
```

can be used to draw the lines. The temporary variable `y` is declared in the loop body so it is local to the loop body. ■

A common error in examples like these is to forget to initialize the boolean expression before entering the loop, or to forget to update it inside the loop. This often results in infinite loops.

7.2.1 Converting a digit string to an integer

Most computer languages provide methods for converting a numerical string to an integer or a floating point number. In Java we have the following static methods.

- **`public int Integer.parseInt(String s)`**

  Static method in the `Integer` class to convert **`s`** to an `int` value and return it.
- **`public long Long.parseLong(String s)`**

  Static method in the `Long` class to convert **`s`** to a `long` value and return it.
- **`public float Float.parseInt(String s)`**

  Static method in the `Float` class to convert **`s`** to a `float` value and return it.
- **`public double Double.parseDouble(String s)`**

  Static method in the `Double` class to convert **`s`** to a `double` value and return it.

Each of these methods throws a `NumberFormatException` if the string `s` is not a valid number of the appropriate type. The classes `Integer`, `Long`, `Double` and `Float` are called **wrapper classes**. They contain useful methods that operate on the corresponding primitive types.

As an example we show how to convert a digit string of the form $s = c_0c_1 \ldots c_{n-1}$ where the c_k are the digit characters to an integer of the form $d = d_0d_1 \ldots d_{n-1}$ where each digit $d_k = c_k -$ '0' is obtained by subtracting the code of the character '0' from the code of c_k. For example, the digits '0' to '9' have codes 48 to 57 so subtracting the code for '0' (48) from each gives the integers 0 to 9. The integer value of the string is accumulated in a loop using the formula

$$d = d_{n-1} + 10(d_{n-2} + \cdots + 10(d_1 + 10d_0)\cdots)$$

A pseudo-code algorithm is given in Figure 7.3.

■ $\boxed{\text{EXAMPLE 7.5}}$ **(Converting a string to an integer)** The following method

```
**ALGORITHM** stringToInt $(c_0, c_1, \ldots, c_{n-1})$
$value \leftarrow 0$
$k \leftarrow 0$
**WHILE** $k < n$ **DO**
    $value \leftarrow (c_k - \text{'0'}) + 10 \times value$
    $k \leftarrow k + 1$
**END WHILE**
**RETURN** $value$
```

Figure 7.3: Algorithm to convert a string to an integer

```
public int stringToInt(String s)
{
   int numDigits = s.length();
   int value = 0;
   int k = 0;
   while (k < numDigits)
   {
      value = (s.charAt(k) - '0') + 10 * value;
      k = k + 1;
   }
   return value;
}
```

gives a Java implementation of the algorithm in Figure 7.3. In Java an automatic typecast is performed to convert the `char` type to the `int` type. The `stringToInt` method is easily tested using the BeanShell workspace and editor. ■

Here is a simple Java class that can be used to test the method in BlueJ.

Class `StringToIntConverter`

book-projects/chapter7/conversion

```
package chapter7.conversion; // remove this line if you're not using packages
/**
 * A class to test the stringToInt method that converts a string to an int.
 */
public class StringToIntConverter
{
   /**
    * convert a string of digits to an int
    * @param s the digit string to convert
    * @return int value of digit string
    */
   public int stringToInt(String s)
   {
      ....
   }
}
```

Here is a runner class that can be used from the command-line or within BlueJ.

Class `StringToIntRunner`

book-projects/chapter7/conversion

```
package chapter7.conversion; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Runner class to test stringToInt method
 */
public class StringToIntRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      StringToIntConverter converter = new StringToIntConverter();
      System.out.println("Enter digit string");
      String digitString = input.nextLine();
      int value = converter.stringToInt(digitString);
      System.out.println("int value is " + value);
   }

   public static void main(String[] args)
   {
      new StringToIntRunner().run();
   }
}
```

7.2.2 Square root algorithm using a while-loop

If we didn’t have the `Math.sqrt` function how could we calculate the square root of a non-negative number? There are many algorithms to do this. A famous one, although not the most efficient, for computing $\sqrt{a}$ is to define the sequence of numbers $x_0, x_1, \ldots, x_n, \ldots$, using the formula

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right), \quad \text{for } n = 1, 2, 3, \ldots$$

This means that if we start with a value for x_0 and substitute $n = 1$ in this formula we get a value for x_1 defined in terms of x_0 by

$$x_1 = \frac{1}{2}\left(x_0 + \frac{a}{x_0}\right)$$

Then we can substitute $n = 2$ to get a value for x_2 defined in terms of x_1 by:

$$x_2 = \frac{1}{2}\left(x_1 + \frac{a}{x_1}\right)$$

```
**ALGORITHM** squareRoot ($a$)
$x_{old} \leftarrow 1$
$x_{new} \leftarrow a$
**WHILE** $|(x_{new} - x_{old})/x_{new}| > 10^{-16}$ **DO**
    $x_{old} \leftarrow x_{new}$
    $x_{new} \leftarrow 0.5 \times (x_{old} + a/x_{old})$
**END WHILE**
**RETURN** $x_{new}$
```

Figure 7.4: Algorithm to compute $\sqrt{a}$.

So if we start with a value for x_0 we can compute the sequence of numbers. It can be shown that these numbers get closer and closer to $\sqrt{a}$, for any starting value $x_0 > 0$.

As an example, compute an approximation to $\sqrt{2}$ using $a = 2$, starting with $x_0 = 1$. Then $x_1 = (1/2)(1+2) = 3/2 = 1.5$, $x_2 = (1/2)(3/2+4/3) \approx 1.46667$, $x_3 \approx 1.41422$, and so on. These numbers get closer and closer to $\sqrt{2}$, which is approximately 1.41421.

A simple pseudo-code algorithm that uses the relative error (see Example 6.9) as a measure of the closeness of successive iterations is given in Figure 7.4.

■ EXAMPLE 7.6 **(Square root method)** The following method

```
public double squareRoot(double a)
{
   double xOld = 1;
   double xNew = a;

   while (Math.abs( (xNew - xOld) / xNew ) > 1E-16 )
   {
      xOld = xNew;
      xNew = 0.5 * (xOld + a / xOld);
   }
   return xNew;
}
```

gives a Java implementation of the algorithm in Figure 7.4. The `squareRoot` method is easily tested using the BeanShell workspace and editor. ■

Here is a simple Java class that can be used to test the method in BlueJ.

Class `SquareRootCalculator`

book-projects/chapter7/square_root

```
package chapter7.square_root; // remove this line if you're not using packages
/**
```

```
 * A simple class to test the square root algorithm.
 */
public class SquareRootCalculator
{
   /**
    * Calculate the square root of a number.
    * @param a the number to take square root of
    * @return the square root of a.
    */
   public double squareRoot(double a)
   {
      ...
   }
}
```

If you want to see the iterations and watch them converge insert the statement

```
      System.out.println(xNew);
```

after the assignment to `xNew` in the while-loop. To check the results you could also print the value of `xNew * xNew` just before the `return` statement to verify how close the result is to the input value `a`, or you can print the value of `Math.sqrt(a)`.

Here is a runner class that can be used from the command-line. It squares the root as a check and also compares the root with `Math.sqrt`.

Class `SquareRootRunner`

book-projects/chapter7/square_root

```
package chapter7.square_root; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * Runner class to test squareRoot method
 */
public class SquareRootRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      SquareRootCalculator calculator = new SquareRootCalculator();
      System.out.println("Enter number");
      double a = input.nextDouble();
      input.nextLine(); // eat end of line
      double root = calculator.squareRoot(a);
      System.out.println("Square root of " + a + " is " + root);
      System.out.println("Square of root is " + root * root);
      System.out.println("Square root using Math.sqrt() is " + Math.sqrt(a));
   }

   public static void main(String[] args)
   {
      new SquareRootRunner().run();
```

```
**ALGORITHM** doublingTime($x, r$)
$monthlyRate \leftarrow r/1200$
$month \leftarrow 0$
$value \leftarrow x$
**WHILE** $value < 2x$ **DO**
    $month \leftarrow month + 1$
    $value \leftarrow value \times (1 + monthlyRate)$
**END WHILE**
**RETURN** $month$
```

Figure 7.5: Pseudo-code algorithm for doubling your money

```
   }
}
```

7.2.3 Double your money problem

Consider the following problem:

> *"How many months does it take to double an initial investment of x dollars if the annual interest rate is r %, and interest is compounded monthly."*

To develop an algorithm we need to know that if you have an amount of money V, and it accumulates interest at a rate r for a period of time, then the interest at the end of the period is rV, and the value of V at the end of the period is $V + rV = V(1+r)$. For our problem the annual rate of $r\%$ percent is converted into the monthly rate $r/100/12$ as a fraction so at the end of a month the amount is $V(1 + r/1200)$ where V is the value at the end of the previous month. A pseudo-code version of the doubling algorithm is shown in Figure 7.5.

■ EXAMPLE 7.7 **(Double your money method)** The following method

```
public int doublingTime(double initialValue, double annualRate)
{
   double value = initialValue;
   double monthlyRate = annualRate / 100.0 / 12.0;
   int month = 0;
   while (value < 2.0 * initialValue)
   {
      month = month + 1;
      value = value * (1.0 + monthlyRate);
   }
   return month;
}
```

gives a Java implementation of the algorithm in Figure 7.5. The `doublingTime` method can be tested using the BeanShell workspace and editor. ■

Here is a simple Java class that can be used to test the method in BlueJ.

Class `DoubleYourMoney`

book-projects/chapter7/money

```
package chapter7.money; // remove this line if you’re not using packages
/**
 * A simple class for the double your money problem.
 */
public class DoubleYourMoney
{
   /**
    * Return the number of months to double your money.
    * @param initialValue the initial investment amount
    * @param annualRate annual interest rate in percent
    * @return the number of months for initial amount to double
    */
   public int doublingTime(double initialValue, double annualRate)
   {
      ...
   }
}
```

Here is a runner class that can be used from the command-line. It also converts the total number of months for doubling into years and months.

Class `DoubleYourMoneyRunner`

book-projects/chapter7/money

```
package chapter7.money; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Runner class to test squareRoot method
 */
public class DoubleYourMoneyRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      DoubleYourMoney calculator = new DoubleYourMoney();
      System.out.println("Enter initial investment amount");
      double amount = input.nextDouble();
      input.nextLine(); // eat end of line
      System.out.println("Enter annual rate in percent");
      double rate = input.nextDouble();
      input.nextLine(); // eat end of line
      int totalMonths = calculator.doublingTime(amount, rate);
      int years = totalMonths / 12;
      int months = totalMonths % 12;
      System.out.println("The amount doubles in "
ALGORITHM factor($n$)
$q \leftarrow n$  // initial quotient
$t \leftarrow 2$  // initial trial factor
WHILE  $t \leq \sqrt{q}$  DO
    IF  $t$ divides $q$  THEN
        Save $t$ as a factor of $q$
        $q \leftarrow q$ div $t$
    ELSE
        $t \leftarrow$ next trial factor
    END IF
END WHILE
Save $q$ as last factor
```

Figure 7.6: Pseudo-code factorization algorithm

```
         + years + " years and " + months + " months");
   }

   public static void main(String[] args)
   {
      new DoubleYourMoneyRunner().run();
   }
}
```

7.2.4 Factorization of an integer

The fundamental theorem of arithmetic states that every integer can be expressed uniquely as a product of prime numbers arranged in increasing order. Recall that a prime number has no factors other than itself and 1 (e.g., 18 is not prime since 6 is a factor but 17 is prime). This means that every integer n can be expressed uniquely as the product

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k}, \text{ where } p_1 < p_2 < \cdots < p_k \text{ are primes}$$

For example

$$\begin{aligned} 140931360 &= 2^5 \times 3^3 \times 5 \times 17 \times 19 \times 101 \\ 140931369 &= 3^2 \times 239 \times 65519 \end{aligned}$$

We can write a method to do this factorization using a while-loop. The largest factor t that an integer q can have is $\sqrt{q}$ so we can use this condition to terminate the while-loop. The algorithm begins by using $t = 2$ as the first trial factor and $q = n$ as the first quotient. If t is a factor of q then we save the factor t and divide it out of q to get the next quotient.

This gives the pseudo-code algorithm in Figure 7.6. To obtain the next trial factor we choose 3 if t is 2 otherwise t is odd and we can choose the next odd factor $t+2$.

To write a Java method for this algorithm we can keep track of factors by appending them to a string as they are obtained so that the output for $n = 140931360$ is expressed as the string

```
<2,2,2,2,2,3,3,3,5,17,19,101>
```

■ EXAMPLE 7.8 **(Factorization method)** The following method

```
public String factor(int n)
{
   int q = n; // initial quotient
   int t = 2; // initial trial factor
   String factors = "<"; // string to hold factors

   // a factor cannot be larger than square root of quotient
   while (t <= q / t)
   {
      if (q % t == 0)
      {
         // t is a factor so append it to string and
         // divide it out of quotient

         factors = factors + t + ",";
         q = q / t;
      }
      else
      {
         // t is not a factor so get the next trial factor.
         // After 2 all trial factors will be odd.

         t = (t == 2) ? 3 : t + 2;
      }
   }
   factors = factors + q + ">";
   return factors;
}
```

implements the algorithm in Figure 7.6 by returning the factors of *n* as a string. The `factor` method can be tested using the BeanShell workspace and editor. ■

Here is a simple Java class that can be used to test the method in BlueJ.

Class `Factorizer`

book-projects/chapter7/factors

```
package chapter7.factors; // remove this line if you're not using packages
/**
 * A class to test the factor method that finds all the prime factors
 * of a number and returns them in a string.
 */
public class Factorizer
```

```
{
   /**
    * Find all the factors of a number.
    * @param n the number to factor
    * @return the string containing the factors
    */
   public String factor(int n)
   {
      ...
   }
}
```

The following class can be used within BlueJ or from the command-line to display the factorization of 10 consecutive integers given the first one.

Class `FactorizerRunner`

book-projects/chapter7/factors

```
package chapter7.factors; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A runner class to test Factorizer by displaying the factorization
 * of all numbers in a given range.
 */
public class FactorizerRunner
{
   /** Factorize 10 numbers starting with a given number
    * @param n the given number
    * @return string of form <f1,f2,...,fn>
    */
   public void displayFactors(int n)
   {
      Factorizer f = new Factorizer();

      int k = n;
      while (k <= n + 9)
      {
         String factors = f.factor(k);
         System.out.println(k + " = " + factors);
         k++;
      }
   }

   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter first value of n");
      int n = input.nextInt();
      input.nextLine(); // eat end of line
      new FactorizerRunner().displayFactors(n);
   }
}
```

Some typical output is

```
java FactorizerRunner
Enter first value of n
140931360
140931360 = <2,2,2,2,2,3,3,3,5,17,19,101>
140931361 = <227,383,1621>
140931362 = <2,11,11,13,44797>
140931363 = <3,31,1515391>
140931364 = <2,2,7,157,32059>
140931365 = <5,571,49363>
140931366 = <2,3,23488561>
140931367 = <353,399239>
140931368 = <2,2,2,17616421>
140931369 = <3,3,239,65519>
```

7.3 Sentinel-controlled while-loops

Let us write a program that read a series of student marks using console input and calculates their average. Assume that the number of marks entered by the user is not known in advance. Since the value of a valid mark ranges from 0 to 100, we can design the program so that it stops asking the user for numbers when the user enters a negative value to indicate the end of the input. This kind of fictitious value, used to indicate the end of the input data, is often called a **sentinel value**. If the first mark entered is negative, then the while-loop is never executed. In this case we assign an average of zero. The following method computes the average mark.

■ EXAMPLE 7.9 **(Sentinel-controlled while-loop)** The following method computes the average of a list of marks and displays it.

```
public void averageMark()
{
   Scanner input = new Scanner(System.in);
   double sum = 0.0;
   int numberOfMarks = 0;
   double mark;

   System.out.println("Enter mark (negative to quit)");
   mark = input.nextDouble();
   input.nextLine(); // eat end of line

   while (mark >= 0.0)
   {
      if (mark <= 100.0)
      {
         sum = sum + mark;
         numberOfMarks = numberOfMarks + 1;
```

```
            }
            else
            {
               System.out.println("Marks > 100 are invalid");
            }
            System.out.println("Enter mark (negative to quit)");
            mark = input.nextDouble();
            input.nextLine(); // eat end of line
         }
         System.out.println("Average mark is " + sum / numberOfMarks);
      }
```

A common mistake in loops like this is to forget to read a new mark at the bottom of the loop before returning to the top again. The result is an infinite loop. ■

7.3.1 `AverageMarkCalculator` class

Here is a simple class to test the method from the command line or from within BlueJ.

Class `AverageMarkCalculator`

book-projects/chapter7/loops

```
package chapter7.loops; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * A class to illustrate a sentinel-controlled while loop
 */
public class AverageMarkCalculator
{
   /**
    * Read marks and compute average mark
    */
   public void averageMark()
   {
      ...
   }

   public static void main(String[] args)
   {
      new AverageMarkCalculator().averageMark();
   }
}
```

7.4 Query-controlled while-loops

In a query-controlled interactive loop the user is asked at the end of every iteration if there is more data to enter. If the answer is yes, another iteration is performed by reading new data and processing it. This kind of loop is needed when there is no sentinel value that can be used. For example, a program that reads an arbitrary series of numbers cannot use any one of them as a sentinel value.

For interactive console input a boolean-valued method, similar to the following one, can be used to test for-loop termination in a query-controlled loop.

```
public boolean moreValues()
{
   System.out.println("Do you want to enter another value [Y/N ?]");
   String reply = input.nextLine();
   return reply.equals("") || reply.toUpperCase().charAt(0) == ’Y’;
}
```

Here we assume that `input` is a `Scanner` object. The method returns `true` if the user enters a blank line or types something that begins with `y` or `Y`, indicating that the user wants to enter another value. Short-circuit evaluation is used for the boolean expression

```
reply.equals("") || reply.toUpperCase().charAt(0) == ’Y’
```

If the user enters a blank line, `reply` is empty and `reply.equals("")` is true, so the right operand of the “or expression” is never evaluated. Therefore, the evaluation of `charAt` is never attempted for an empty string.

Using this method, the query-controlled while-loop can be written as

```
while (moreValues())
{
   // read a value here
   // process the value here
}
```

This loop works even if there are no values to enter.

7.4.1 **`BankAccount`** example

We want to write a class that uses a query-controlled while-loop to find the bank account with the maximum balance from a list of accounts entered using the console. The heart of the class is the method

```
public BankAccount findMaxBalance()
{
   BankAccount maxAccount = readAccount();
   while (moreAccounts())
   {
      BankAccount next = readAccount();
      if (next.getBalance() > maxAccount.getBalance())
      {
         maxAccount = next;
      }
```

```
         }
         return maxAccount;
      }
```

that finds the account with the maximum balance and returns a reference to it. This reference starts out as a reference to the first account and each time an account with a larger balance is read the `maxAccount` reference is updated to refer to this account. When the method exits all the account objects, except the one with the maximum balance referenced by `maxAccount`, will be orphans (no references to them) so they will be garbage collected.

Here is a Java class using this method that can be run within BlueJ or from the command-line.

Class `MaxBalanceCalculator`

book-projects/chapter7/loops

```
package chapter7.loops; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * A class to illustrate the query-controlled while loop by reading
 * a list of accounts from the console and finding the one that
 * has the maximum balance.
 */
public class MaxBalanceCalculator
{
   Scanner input = new Scanner(System.in);

   /**
    * Read accounts from console
    * @return reference to account having the maximum balance
    */
   public BankAccount findMaxBalance()
   {
      BankAccount maxAccount = readAccount();
      while (moreAccounts())
      {
         BankAccount next = readAccount();
         if (next.getBalance() > maxAccount.getBalance())
         {
            maxAccount = next;
         }
      }
      return maxAccount;
   }

   private boolean moreAccounts()
   {
      System.out.println("Do you want to enter another account [Y/N ?]");
      String reply = input.nextLine();
      return reply.equals("") || reply.toUpperCase().charAt(0) == ’Y’;
   }
   private BankAccount readAccount()
   {
      System.out.println("Enter account number");
      int number = input.nextInt();
      input.nextLine(); // eat end of line

      System.out.println("Enter owner name");
      String name = input.nextLine();

      System.out.println("Enter balance");
      double balance = input.nextDouble();
      input.nextLine(); // eat end of line
      return new BankAccount(number, name, balance);
   }

   public static void main(String[] args)
   {
      MaxBalanceCalculator calc = new MaxBalanceCalculator();
      System.out.println("Account with maximum balance is " + calc.findMaxBalance());
   }
}
```

In BlueJ use "Add class from files" to add the `BankAccount` class from the `custom-classes` project. The class also uses two private methods to read an account and do the query-controlled test for another account.

7.5 Do-while statement (do-while loop)

In the while-loop the test for loop termination is always done at the top of the loop. There are cases when we would like to do the test at the bottom of the loop. Several variations occur in programming languages. In pseudo-code we could use the structure

```
REPEAT
    Statements
WHILE BooleanExpression
```

which repeats *Statements* while the boolean expression is true. Alternatively we could use the negated form

```
REPEAT
    Statements
UNTIL BooleanExpression
```

which repeats *Statements* until the boolean expression is true (or equivalently, while it is not true). In either case, unlike the while-loop, the statements in the loop are always executed at least once. In practice the do-while loop is not as common as the while-loop.

Languages such as Pascal and Modula-2 have a repeat-until statement. Others, C, C++ and Java for example, have a repeat-while statement (called do-while) shown in Figure 7.7. The do-while statement needs a semi-colon at the end of the while-part.

A flowchart for the execution of the do-while statement is shown in Figure 7.8.

```
do
{
    Statements
}
while ( BooleanExpression );
```

Figure 7.7: A template for the do-while statement

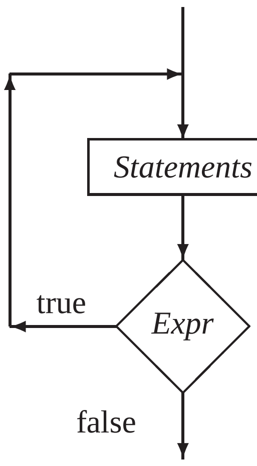


Figure 7.8: A flowchart for the execution of a do-while statement

■ EXAMPLE 7.10 **(Counting up with a do-while loop)** The statements

```
int count = 1;
do
{
   System.out.print(count + " ");
   count = count + 1;
}
while (count <= 10);
```

display `1 2 3 4 5 6 7 8 9 10`. After `10` is displayed `count` is incremented to `11`, the boolean expression `count <= 10` becomes false and the loop exits. Try this example using the BeanShell workspace and editor. Also choose “Capture System in/out/err” from the “File” menu. ■

■ EXAMPLE 7.11 **(Counting down with a do-while loop)** The statements

```
int count = 10;
do
{
   System.out.print(count + " ");
```

```
        count = count - 1;
    }
    while (count > 0);
```

display `10 9 8 7 6 5 4 3 2 1`. After `1` is displayed `count` is decremented to `0`, the boolean expression `count > 0` becomes false and the loop exits. Try this example using the BeanShell workspace and editor. Also choose “Capture System in/out/err” from the “File” menu. ■

7.6 General loop structures

The while-loop makes the test at the top of the loop and the do-while loop makes it at the bottom of the loop. It is possible to generalize and make the test somewhere in the middle of the loop. In pseudo-code we could invent a general loop-structure such as

```
LOOP
    StatementsA
    IF BooleanExpression THEN EXIT
    StatementsB
END LOOP
```

Here an **IF** statement with a special **EXIT** statement is used to exit the loop somewhere in the middle if *BooleanExpression* is true. The while-loop is the special case that there is no *StatementA* block and the repeat-until loop (do-while in negated form) is the special case when there is no *StatementB* block.

General loops like this can be hard to read since there could be several if-exit statements in the loop. If possible you should always try to write your loops in the while or do-while forms or using the for-statement discussed next.

Java does not have a **LOOP** statement but it has a `break` statement corresponding to **EXIT** so we can write a general loop using the while-loop shown in Figure 7.9.

```
while(true)
{
    StatementsA
    if ( BooleanExpression ) break;
    StatementsB
}
```

Figure 7.9: A template for a general loop structure.

7.7 For-statement (for-loop)

The for-statement (for-loop) is the last of the three repetition statements. It is used to repeat one or more statements a fixed number of times, determined in advance, either as a constant or as the value of an expression.

7.7.1 Pseudo-code for-loops for counting in steps

In pseudo-code the for-statement can be expressed as

```
FOR $k \leftarrow start$ TO $end$ BY $step$ DO
    Statements
END FOR
```

or

```
FOR $k \leftarrow start$ TO $end$ BY $-step$ DO
    Statements
END FOR
```

where we assume that $step > 0$. In the first case the **BY** part can be omitted in the most common case that *step* is 1.

Here k is called the loop variable (or the loop counter). It is initialized to the value *start* and is incremented or decremented automatically each time the *Statements* block is executed. The value of *step* determines how much is added to or subtracted from the loop counter each time the block is executed. The two cases define an upward counting loop and a downward counting loop and can be described as follows.

- **BY** *step*: In this case the values of k are $start$, $start + step$, $start + 2\,step$, and so on, increasing and ending with the last value such that $k \le end$. If $start > end$ the for-loop is ignored.
- **BY** $-step$: In this case the values of k are $start$, $start - step$, $start - 2\,step$, and so on, decreasing and ending with the last value such that $k \ge end$. If $start < end$ the for-loop is ignored.

The important special cases occur when $step = 1$ for which the loops count upward from *start* to *end* in steps of 1 in the first case or downward from *start* to *end* in steps of 1 in the second case.

In Java the for-statement is needlessly complicated (to please C and C++ programmers) but we can easily use it to model the pseudo-code versions even though there are more general versions. A template is shown in Figure 7.10. As in the case of the while and do-while statements, the body of the for-statement must be enclosed in braces if it contains more than one statement.

The *Initialization* part normally involves a numeric variable declaration with initialization. This variable is the loop counter.

The *Test* part is a boolean expression which depends on the loop counter. If the value of the condition is true, the statements in the body of the loop are executed.

If the statements in the loop are executed (because *Test* was true initially) the *Update* statement is then executed. It's purpose its to modify the loop counter (increase or decrease it). Then *Test* is

```
for ( Initialization ; Test ; Update )
{
   Statements
}
```

Figure 7.10: A template for the for-statement.

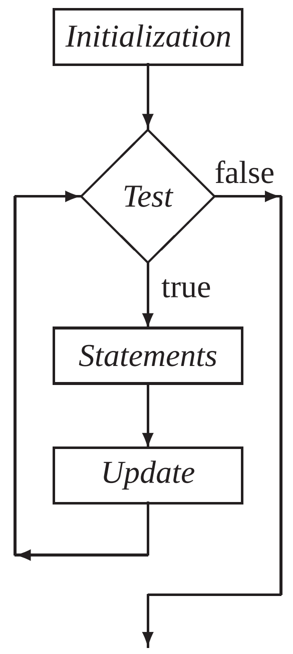


Figure 7.11: A flowchart for the execution of a for-statement

evaluated again to see if the loop statements should be executed again. Eventually *Update* should modify the counter so that *Test* will be false and terminate the loop.

The flowchart for the for-loop is shown in Figure 7.11. The loop counter is commonly a variable of type `int` although a variable of type `double` can also be used.

7.7.2 For-loops for counting in steps

The first case of the pseudo-code for-statement can be expressed in Java as

```
for (int k = start; k <= end; k = k + step)
{
   // loop statements
}
```

If `step` is positive the values of `k` count up from `start`, in steps of `step`, ending at the largest value of `k` less than or equal to `end`. The initialization part is just an initialized variable declaration statement for the loop counter variable which is local to the for-loop The test is a boolean expression which will be true until the value of `k` exceeds `end`. The update part, which can also be expressed as `k += step`, adds the value of `step` to `k` each time through the loop.

Similarly, the second case of the pseudo-code for-statement can be expressed as

```
for (int k = start; k >= end; k = k - step)
{
   // loop statements
}
```

If `step` is positive the values of `k` count down from `start`, in steps of `step`, ending in the smallest value of `k` greater than or equal to `end`. The test is a boolean expression which will be true until the value of `k` is below `end`.

■ EXAMPLE 7.12 **(Counting up from 1 to 10)** The for-statement

```
for (int k = 1; k <= 10; k++)
{
   System.out.print(k + " ");
}
```

displays `1 2 3 4 5 6 7 8 9 10`. It is equivalent to the while-loop in Example 7.1. The body of the loop will be repeated 10 times since `k` takes on the values 1 to 10. Here, the loop counter `k` is incremented by 1 every time the body of the loop is executed. It is common to use `k++` in the update part of the for-loop, instead of the equivalent assignment `k = k + 1`. ■

■ EXAMPLE 7.13 **(Counting down from 10 to 1)** Similarly, the for-statement

```
for (int k = 10; k >= 1; k--)
{
   System.out.print(k + " ");
}
```

displays `10 9 8 7 6 5 4 3 2 1`. It is equivalent to the while-loop in Example 7.2. ■

■ EXAMPLE 7.14 **(Counting down by 3)** The for-statement

```
for (int k = 10 ; k > 0 ; k = k - 3)
{
   System.out.print(k + " ");
}
```

displays `10 7 4 1`. Each time the loop executes `k` is decremented by 3. Printing stops at 1 since the next value of `k` would be `-2` which makes `k > 0` is false. ■

■ EXAMPLE 7.15 **(Drawing some horizontal lines)** The statements in Example 7.4 that draw 10 horizontal lines can be more easily expressed as

```
for (int k = 0; k <= 9; k++)
{
   double y = 10.0 + 20.0*k; // y coordinate of line k
   g2D.draw(new Line2D.Double(10,y,200,y));
}
```

using the for-loop. ■

■ EXAMPLE 7.16 **(Computing $1+2+\cdots+n$)** Given a value for `n` the for-loop

```
int sum = 0;
for (int k = 1; k <= n; k++)
{
   sum = sum + k;
}
```

computes the sum of the first *n* integers. ■

We shall see in the remainder of this Chapter and in Chapter 8 that the for-loop has many applications.

7.8 Computing factorials

We can use a for-loop to compute $n!$. For values of type `int` we can do this before overflow only if $0 \le n \le 12$. This range can be extended using the `long` data type (see Exercise 7.1). We can also calculate large factorials using objects from the `BigInteger` class in package `java.math` that represent arbitrarily large integers and perform arithmetic operations on them limited only by the amount of memory available.

7.8.1 Computing the factorial of an integer

For a non-negative integer *n*, the factorial of *n*, denoted by $n!$, is defined by

$$n! = \begin{cases} 1 \times 2 \times \cdots \times n, & n > 0 \\ 1, & n = 0 \end{cases}$$

We can write a method that uses a for-loop to compute $n!$. The algorithm is simple: initialize a `product` variable to 1, multiply it by 2, multiply it by 3, and so on, until it is multiplied by *n*. There will be $n-1$ multiplications:

■ EXAMPLE 7.17 **(Method for $n!$)** The following method

```
int factorial(int n)
{
   int product = 1;
   for (int k = 2; k <= n; k++)
   {
      product = product * k;
   }
   return product;
}
```

calculates $n!$ and returns it. The `product` variable successively takes on the values `2 = 1*2`, `6 = 2*3`, `24 = 6*4`, etc. For $n = 0$, or $n = 1$, the loop will not be executed even once, because `k <= n` is false if `k` is 2, so the final value of `product` will be 1. This agrees with the definition of $n!$. The method can easily be tested using the BeanShell editor and workspace. In particular you can verify that n must be in the range $0 \leq n \leq 12$. ■

Here is a simple tester class that can be used in BlueJ using a modified method that throws an exception if n is outside the range $0 \leq n \leq 12$.

Class `FactorialCalculator`

book-projects/chapter7/factorial

```
package chapter7.factorial; // remove this line if you're not using packages
/**
 * A simple class to test the factorial method.
 */
public class FactorialCalculator
{
   /**
    * Calculate n!.
    * @param n value for n!
    * @return n!
    * @throws IllegalArgumentExeception if n is outside the range
    * 0 &lt;= n &lt;= 12.
    */
   public int factorial(int n)
   {
      if (n < 0 || n > 12)
      {
         throw new IllegalArgumentException("n! is only defined for n = 0..12");
      }
      int product = 1;
      for (int k = 2; k <= n; k++)
      {
         product = product * k;
      }
      return product;
   }
}
```

For command-line testing outside BlueJ the following class can be used.

Class `FactorialRunner`

book-projects/chapter7/factorial

```
package chapter7.factorial; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing factorial method from commmand line
 */
public class FactorialRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      FactorialCalculator calc = new FactorialCalculator();
      System.out.println("Enter value of n");
      int n = input.nextInt();
      System.out.println(n + "! = " + calc.factorial(n));
   }

   public static void main(String[] args)
   {
      new FactorialRunner().run();
   }
}
```

7.8.2 Computing factorials using the `BigInteger` class

Java has a `BigInteger` class in the `java.math` package which represents arbitrarily large integers and arithmetic operations limited only by the amount of available memory. We can use it to calculate large factorials. Each integer is represented as a `BigInteger` object. To convert a normal `int` or `long` integer to a `BigInteger` object there is the static method called `valueOf` with prototype

```
public static BigInteger valueOf(long val)
```

For example,

```
BigInteger bigI = BigInteger.valueOf(1);
```

converts the integer value 1 to a big integer. There is also a constructor that takes a string as an argument and uses it to construct a big integer object. It has the prototype

```
public BigInteger(String val)
```

For example,

```
BigInteger bigI = new BigInteger("1111111111111111111111111111111111111111111");
```

converts the given number string (too big even for type `long`) to a big integer.

There are `add`, `subtract`, `multiply` and `divide` methods for big integers. The `multiply` method has prototype

```
public BigInteger multiply(BigInteger val)
```

For example, the following statements multiply two big integer objects `b1` and `b2` to produce a new object which is assigned to `b3`.

```
BigInteger b3 = b1.multiply(b2);
```

Finally, we need a way to convert a big integer result to a string, so that we can display it. To do this there is a `toString` method with prototype

```
public String toString()
```

There are many other methods in the `BigInteger` class (see Java class documentation) but we don't need them to compute big factorials.

To define and initialize the `product` variable to 1 we can use the static `valueOf` method:

```
BigInteger product = BigInteger.valueOf(1);
```

To obtain a big integer object representing the integer loop counter `k` we can use the `valueOf` method:

```
BigInteger bigK = BigInteger.valueOf(k);
```

We don't need to make the loop counter `k` into a big integer. To multiply `product` by `bigK` in the loop use

```
product = product.multiply(bigK);
```

■ EXAMPLE 7.18 **(Big integer method for *n*!)** The following method

```
BigInteger bigFactorial(int n)
{
   BigInteger product = BigInteger.valueOf(1);
   for (int k = 2; k <= n; k++)
   {
      BigInteger bigK = BigInteger.valueOf(k);
      product = product.multiply(bigK);
   }
   return product;
}
```

is the `BigInteger` version of *n*!. ■

To test this method we need a way to display the large answers. For example, 200! has 375 digits (`toString` returns a string of 375 characters). To do this we can break the string into blocks of a given number of characters per line using the method

```
private void displayLongString(String s, int width)
{
   int length = s.length();
   int numberOfLines = length / width;

   for (int k = 0; k < numberOfLines; k++)
   {
```

```
            System.out.println(s.substring(k * width,  (k+1) * width));
         }
         if (length % width != 0) // display a final partial line
            System.out.println(s.substring(numberOfLines * width));
      }
```

which displays `width` characters of `s` per line by extracting substrings.

Here is a class that uses this method to test the `bigFactorial` method.

Class `BigFactorialCalculator`

book-projects/chapter7/factorial

```
package chapter7.factorial; // remove this line if you're not using packages
import java.math.BigInteger;
/**
 *  Compute n factorial using BigInteger arithmetic.
 */
public class BigFactorialCalculator
{
   /**
    * Compute factorials using BigInteger arithmetic and display them.
    * @param n the value whose factorial is to be calculated.
    */
   public void displayFactorial(int n)
   {
      String s = bigFactorial(n).toString();
      System.out.println("Number of digits is " + s.length());
      System.out.println(n + "! = ");
      displayLongString(s, 60);
   }

  /* Compute n factorial using BigInteger arithmetic
   */
   private BigInteger bigFactorial(int n)
   {
      ...
   }

   /* Display the string s, width characters per line
    */
   private void displayLongString(String s, int width)
   {
      ...
   }
}
```

For command-line testing the following class can be used.

Class `BigFactorialRunner`

book-projects/chapter7/factorial

```
package chapter7.factorial; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing big factorial method from command line
 */
public class BigFactorialRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      BigFactorialCalculator calc = new BigFactorialCalculator();
      System.out.println("Enter value of n");
      int n = input.nextInt();
      calc.displayFactorial(n);
   }

   public static void main(String[] args)
   {
      new BigFactorialRunner().run();
   }
}
```

The output for 200! is

```
java BigFactorialRunner
Enter value of n
200
200! =
788657867364790503552363213932185062295135977687173263294742
533244359449963403342920304284011984623904177212138919638830
257642790242637105061926624952829931113462857270763317237396
988943922445621451664240254033291864131227428294853277524242
407573903240321257405579568660226031904170324062351700858796
178922222789623703897374720000000000000000000000000000000000
000000000000000
```

7.9 Expressing the for-loop as a while-loop

The for-loop shown in Figure 7.10 can be expressed as the while-loop shown in Figure 7.12, but the for-loop is simpler in these cases.

■ EXAMPLE 7.19 **(Comparing the for and while-loops)** The for-loop and while-loop for calculating factorials are

Using a for-loop

```
product = 1;
for (int k = 2; k <= n; k++)
{
   product = product * k;
}
```

Using a while-loop

```
product = 1;
int k = 2;
while (k <= n)
{
   product = product * k;
```

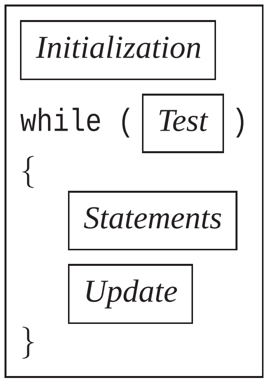


Figure 7.12: A for-loop expressed as a while-loop

```
        k++;
    }
```

Note that the initialization of `k` to 2 has to be done before the while-statement. Also, the update statement `k++` is in the body of the while-loop. Since we know that the loop will execute `n - 2` times the for-loop is more appropriate and easier to write. ■

7.10 Loan repayment table

We want to write a program that solves the following problem

> "*Given the amount of a loan (principal), the number of years to pay back the loan, the number of payments per year, and the annual interest rate, produce a loan repayment table for each payment period showing the principal repaid and the principal remaining.*"

We need some financial mathematics to solve this problem. Let us define the following quantities

A	the amount of the loan (principal)
y	the number of years to pay back the loan
m	the number of payments per year (periods per year)
j	the annual interest rate as a decimal number

Using these quantities we need to compute the following quantities.

$$\begin{aligned} n &= my, \quad \text{the total number of payments,} \\ i &= \frac{j}{m}, \quad \text{the interest rate per payment period as a decimal number,} \\ R &= \frac{A}{\mathrm{a}(n,i)}, \quad \text{the payment made at the end of each payment period,} \end{aligned}$$

where

$$\mathrm{a}(n,i) = \frac{1-(1+i)^{-n}}{i}$$

Payment Number	Periodic Repayment	Payment of interest	Principal Repaid	Remaining Principal
				A
1	R	$I_1 = iA$	$P_1 = R - iA$	$A_1 = A - P_1$
2	R	$I_2 = iA_1$	$P_2 = R - iA_1$	$A_2 = A_1 - P_2$
3	R	$I_3 = iA_2$	$P_3 = R - iA_2$	$A_3 = A_2 - P_3$
...	...	...	...	...

Table 7.1: A loan repayment table

```
ALGORITHM LoanRepayment(loanAmount, years, paymentsPerYear, annualRate)
n ← paymentsPerYear × years
i ← annualRate / paymentsPerYear
payment ← loanAmount / a(n, i)
principalRemaining ← loanAmount
FOR paymentNumber ← 1 TO n DO
    interest ← principalRemaining × i
    principalRepaid ← payment − interest
    principalRemaining ← principalRemaining − principalRepaid
    OUTPUT paymentNumber, payment, interest,
        principalRepaid, principalRemaining
END FOR
```

Figure 7.13: Pseudo-code loan repayment algorithm

The loan repayment table has the form shown in Table 7.1. The following properties of the table can be used as checks

1. $P_1, P_2, P_3, \ldots,$ satisfy $P_2/P_1 = P_3/P_2 = \cdots = 1+i$
2. In each row the entries in columns 3 and 4 sum to R
3. The total of column 2 is the total amount paid
4. The total of column 3 is the total interest paid
5. The total of column 4 is A, the amount of the loan
6. The entry in column 5 should be 0 after n payments

A pseudo-code algorithm for producing the table is shown in Figure 7.13.

7.10.1 Right justifying numbers in a field of given width

We want to produce a nicely formatted table that looks like

```
Payment     Payment    Interest   Principal   Principal
 Number                    Paid      Repaid   Remaining
```

```
-------     -------    --------   ---------   ---------
      1     1295.05      500.00      795.05     9204.95
      2     1295.05      460.25      834.80     8370.16
    ...         ...         ...         ...         ...
      9     1295.05      120.40     1174.64     1233.38
     10     1295.05       61.67     1233.38        0.00
```

with the columns right-justified in fields and numbers displayed with two digits after the decimal point. We can use the `String.format` method from Chapter 4.2.5 to do this. The format code `%7d` can be used to format the first column of integers and the code `%12.2f` can be used to format the remaining fields.

7.10.2 `StringBuilder` class

We want to make our class general so we do not produce the table using `System.out.print` statements. Instead we will return the entire table as a big formatted string. That way when we discuss graphical user interfaces (GUI's) in a later Chapter, where `System.out.print` has no meaning, we can use our loan repayment class unchanged.

Since there are a lot of string manipulations it is more appropriate to use the `StringBuilder` class in package `java.lang`. This is a mutable version of the `String` class that is more efficient to use when performing a lot of string manipulations. The constructor and method prototypes from this class that we need are

- **`public StringBuilder(int size)`**

  Construct an empty string buffer with space for `size` characters initially. The size will expand as needed.

- **`public void append(String s)`**

  Append the given string `s` to the end of the buffer (like `+` for string concatenation). There are several versions of `append` that have `int` and `double` and other types of arguments. In any case the argument is converted to a string and appended to the string buffer.

- **`public String toString()`**

  Convert the string buffer to a `String` object. We usually do this when we are finished creating the string buffer.

7.10.3 Loan repayment table class

The public interface of our class is

```
public class LoanRepaymentTable
{
   public LoanRepaymentTable(double a, int y, int p, double r) {...}
   public String toString() {...}
}
```

Here `a` is the amount of the loan, `y` is the number of years for repayment, `p` is the number of payments per year and `r` is the annual rate in percent. The `toString` method will return the table as one big string. The complete class is

Class `LoanRepaymentTable`

book-projects/chapter7/loan_repayment

```
package chapter7.loan_repayment; // remove this line if you’re not using packages
/**
 * A class to compute a loan repayment table, given the loan amount, the
 * number of years to repay the loan, the number of payments made per year,
 * and the annual interest rate in percent.
 */
public class LoanRepaymentTable
{
   private double loanAmount; // initial amount of the loan
   private int years; // years to pay back the loan
   private int paymentsPerYear;
   private double annualRate;  // as a fraction
   private String table; // the loan repayment table

   /**
    * Construct a loan repayment table.
    * @param a the given amount of the loan
    * @param y the number of years to pay it back
    * @param p the number of payments per year
    * @param r the annual interest rate in percent
    */
   public LoanRepaymentTable(double a, int y, int p, double r)
   {
      loanAmount = a;
      years = y;
      paymentsPerYear = p;
      annualRate = r / 100.0; // convert percent to a fraction
      computeTable();
   }

   /**
    * Return the loan repayment table.
    * @return the loan repayment table.
    */
   public String toString()
   {
      return table;
   }

   /* Construct the entire table as one big string that
      contains newlines to break the table into lines
   */
   private void computeTable()
   {
      int n = paymentsPerYear * years;
      double i = annualRate / paymentsPerYear;
      double payment = loanAmount / a(n,i);
      double principalRemaining = loanAmount;
      double interest;
      double principalRepaid;

      // append headings

      StringBuilder buffer = new StringBuilder(1000);

      buffer.append("Payment     Payment    Interest   Principal   Principal\n");
      buffer.append(" Number                    Paid      Repaid   Remaining\n");
      buffer.append("-------     -------    --------   ---------   ---------\n");

      // Calculate table and append rows to buffer

      double totalInterestPaid = 0.0;
      double totalPrincipalPaid = 0.0;
      for (int paymentNumber = 1; paymentNumber <= n; paymentNumber++)
      {
         interest = principalRemaining * i;
         principalRepaid = payment - interest;
         principalRemaining = principalRemaining - principalRepaid;

         buffer.append(String.format("%7d", paymentNumber));
         buffer.append(String.format("%12.2f", payment));
         buffer.append(String.format("%12.2f", interest));
         buffer.append(String.format("%12.2f", principalRepaid));
         buffer.append(String.format("%12.2f", principalRemaining));
         buffer.append("\n");

         totalInterestPaid += interest;
         totalPrincipalPaid += principalRepaid;
      }

      double totalLoanCost = totalInterestPaid + totalPrincipalPaid;
      buffer.append("\n");
      buffer.append("Total interest paid is " +
         String.format("%.2f", totalInterestPaid) + "\n");
      buffer.append("Total premium paid is " +
         String.format("%.2f", totalPrincipalPaid) + "\n");
      buffer.append("Total cost of loan is " +
         String.format("%.2f", totalLoanCost) + "\n");
      table = buffer.toString();
   }

   private double a(int n, double i)
   {
      return (1.0 - Math.pow(1.0 + i, -n)) / i;
   }
}
```

Most of the work is done by the `computeTable` method which is called by the constructor. It initializes a `StringBuilder` object called `buffer` and appends rows of the table and new line characters. When the calculations are complete the buffer is returned from the method as the instance data field `table` which can be obtained using the `toString` method.

7.10.4 Console user interface

Here is a runner class that can be used in `BlueJ` and from the command line.

Class `LoanRepaymentTableRunner`

book-projects/chapter7/loan_repayment

```
package chapter7.loan_repayment; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Class for running LoanRepaymentTable from console.
 */
public class LoanRepaymentTableRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter loan amount");
      double a = input.nextDouble();
      input.nextLine();
      System.out.println("Enter number of years");
      int y = input.nextInt();
      input.nextLine();
      System.out.println("Enter number of payments per year");
      int p = input.nextInt();
      input.nextLine();
      System.out.println("Enter annual interest rate in percent");
      double r = input.nextDouble();
      input.nextLine();

      LoanRepaymentTable table = new LoanRepaymentTable(a,y,p,r);
      System.out.println(table);
   }

   public static void main(String[] args)
   {
      new LoanRepaymentTableRunner().run();
   }
}
```

The entire table is displayed in the console or terminal window by the single statement

```
System.out.println(table);
```

The console output for a $10,000 loan, at 10% per year, for a period of 5 years, with payments twice a year is

```
java LoanRepaymentTableRunner
Enter loan amount
10000
Enter number of years
5
Enter number of payments per year
2
Enter annual interest rate in percent
10
Payment     Payment    Interest   Principal   Principal
 Number                    Paid      Repaid   Remaining
-------     -------    --------   ---------   ---------
      1     1295.05      500.00      795.05     9204.95
      2     1295.05      460.25      834.80     8370.16
      3     1295.05      418.51      876.54     7493.62
      4     1295.05      374.68      920.36     6573.25
      5     1295.05      328.66      966.38     5606.87
      6     1295.05      280.34     1014.70     4592.17
      7     1295.05      229.61     1065.44     3526.73
      8     1295.05      176.34     1118.71     2408.02
      9     1295.05      120.40     1174.64     1233.38
     10     1295.05       61.67     1233.38        0.00

Total interest paid is 2950.46
Total premium paid is 10000.00
Total cost of loan is 12950.46
```

7.11 Nested loops

One of the statements in the body of a loop could be another loop statement. We say that the inner loop is nested within the outer loop. The inner loop will be executed for each iteration of the outer one. Nested for-loops are quite common for processing data which has a two-dimensional representation as a number of rows and columns. We consider several simple examples:

■ EXAMPLE 7.20 **(5 rows of 10 circles)** Suppose we want to draw the grid of circles shown in Figure 7.14. Assuming that the radius of each circle is 20, then the top left corner coordinates of the bounding boxes are `(40 * column, 40 * row)` where `row` goes from 0 to 4 and `column` goes from 0 to 9. The width of each box is 40. Therefore, the following nested loop draws the circles, assuming that `g2D` is a `Graphics2D` reference.

```
double size = 40.0;
for (int row = 0; row <= 4; row++)
{
   double yTopLeft = size * row;
   for (int column = 0; column <= 9; column++)
   {
      double xTopLeft = size * column;
      g2D.draw(new Ellipse2D.Double(xTopLeft, yTopLeft, size, size));
   }
```

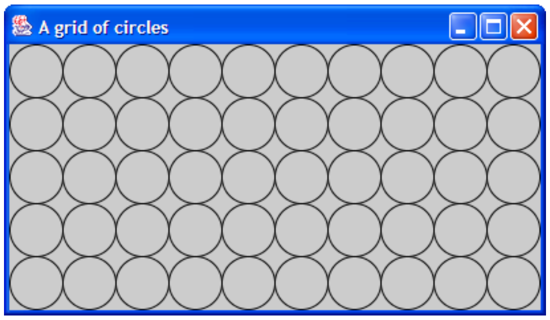


Figure 7.14: 5 rows of 10 circles touching each other

```
   }
```

For each value of `row`, the inner loop draws an entire row of 10 circles indexed by the value of `column`. ■

■ EXAMPLE 7.21 **(Square pattern)** You can experiment with nested loops using console output. For example, the nested loop statement

```
   for (int row = 1; row <= 4; row++)
   {
      for (int column = 1; column <= 10; column++)
      {
         System.out.print("*");
      }
      System.out.println();
   }
```

displays the rectangular pattern

```
   **********
   **********
   **********
   **********
```

with 4 rows of 10 asterisks. Try this example using the BeanShell workspace and editor. ■

■ EXAMPLE 7.22 **(Triangular pattern)** In Example 7.20 and Example 7.21 the inner loop index `column` did not depend on the outer loop index `row`. In the nested loop

```
   for (int row = 1; row <= 4; row++)
   {
      for (int column = 1; column <= row; column++)
      {
         System.out.print("*");
      }
      System.out.println();
   }
```

the inner loop index depends on the outer one and the following triangular pattern is displayed.

```
*
**
***
****
```

For the first iteration of the outer loop (`row = 1`), the inner loop index goes from 1 to 1 so one asterisk is displayed. For the second iteration of the outer loop (`row = 2`), the inner loop index goes from 1 to 2, so two asterisks are displayed. Thus, each row contains one more asterisk than the preceding one. The total number of asterisks displayed is $1+2+\cdots+n = n(n+1)/2$, which is 10 in this case. Try this using the BeanShell workspace and editor. ■

■ EXAMPLE 7.23 **(Doubly-nested loop for computing powers)** The following loop structure computes the second to fifth powers of the numbers one 1 to 10. For a given n defining a row of the table the row contains the numbers n, n^2, n^3, n^4 and n^5. Thus n is the outer loop row index and p, the power, is the inner loop column index. The double loop is given by

```
int val;
for (int n = 1; n <= 10; n++)
{
   System.out.printf("%5d", n);

   val = n;
   for (int p = 2; p <= 5; p++)
   {
      val = val * n;
      System.out.printf("%8d", val);
   }
   System.out.println();
}
```

The output is

```
    1       1       1       1       1
    2       4       8      16      32
    3       9      27      81     243
    4      16      64     256    1024
    5      25     125     625    3125
    6      36     216    1296    7776
    7      49     343    2401   16807
    8      64     512    4096   32768
    9      81     729    6561   59049
   10     100    1000   10000  100000
```

For row n each value in the inner loop is obtained from the one to its left by multiplying by n. We never need to use the `Math.pow` method. ■

7.11.1 Investment table

Let us write a program to print a table showing the value of an investment for different interest rates and different numbers of years. More specifically,

> "*Given an initial investment amount, compute a future value table for different rates from a minimum rate of 4% to a maximum rate of 10% in steps of 0.5%, and for an investment time of 5 to 30 years in steps of 5 years. The rows of the table correspond to the rates, and the columns correspond to the number of years.*"

For example, if the initial investment is $1000 the following table shows the value of the investment, with some rows omitted:

```
 RATE    5 YEARS   10 YEARS   15 YEARS   20 YEARS   25 YEARS   30 YEARS
 4.00    1221.00    1490.83    1820.30    2222.58    2713.77    3313.50
 4.50    1251.80    1566.99    1961.56    2455.47    3073.74    3847.70
 ....    .......    .......    .......    .......   ........   ........
 9.50    1605.01    2576.06    4134.59    6636.06   10650.94   17094.86
10.00    1645.31    2707.04    4453.92    7328.07   12056.94   19837.40
```

For example, at 4.5% an investment of $1000 is worth $3073.74 after 25 years. This table can be produced by a nested for-loop. The outer loop goes over the rows of the table, and the inner loop goes over the columns. We can generalize and use variables for the investment amount, the minimum rate, the maximum rate, the rate step from one row to the next, the minimum number of years, the maximum number of years, and the year step from one column to the next. The nested loop structure has the form

```
double small = 0.00001;
for (double rate = minRate; rate <= maxRate + small; rate += rateStep)
{
   ...
   for (int years = minYears; years <= maxYears; years += yearStep)
   {
      ...
   }
   ...
}
```

The outer loop goes over the rows of the table and the inner loop goes across the columns in a row. We add a small constant to `maxRate` in case there is roundoff error to force `maxRate` to be reached.

The formula for the future value F of an amount A, with a yearly rate of r percent, compounded monthly, for n years is

$$F = A\left(1+\frac{r}{1200}\right)^{12n}$$

We can use this formula to write the following method for calculating the future value.

```
private double futureValue(double presentValue, double yearlyRate, int years)
```

```
      {
         double monthlyRate = yearlyRate / 100.0 / 12.0;
         return presentValue * Math.pow(1.0 + monthlyRate, 12 * years);
      }
```

As in the `LoanRepaymentTable` class we use a `StringBuilder` to accumulate the table as a string and we use the `String.format` method to line everything up in columns. The complete class is given by

Class `InvestmentTable`

book-projects/chapter7/investment

```
package chapter7.investment; // remove this line if you're not using packages
/**
 * A class that produces an investment table for a given initial investment.
 */
public class InvestmentTable
{
   private double minRate, maxRate, rateStep; // range and step for rows
   private int minYears, maxYears, yearStep; // range and step for columns
   private double initialValue; // initial value of investment
   private String table; // the investment table

   /**
    * Construct table for given initial investment, rate range, and year range.
    * @param minRate rate (percent per year) for first table row
    * @param maxRate rate (percent per year) for last table row
    * @param rateStep step size in percent between table rows
    * @param minYears number of years for first column
    * @param maxYears number of years for last column
    * @param yearStep step size in years between table columns
    * @param initialValue initial value of the investment
    */
   public InvestmentTable(double minRate, double rateStep, double maxRate,
      int minYears, int yearStep, int maxYears, double initialValue)
   {
      this.minRate = minRate;
      this.maxRate = maxRate;
      this.rateStep = rateStep;
      this.minYears = minYears;
      this.maxYears = maxYears;
      this.yearStep = yearStep;
      this.initialValue = initialValue;
      computeTable();
   }

   /**
      Return the investment table.
      @return the table
   */
   public String toString()
   {
      return table;
   }

   private void computeTable()
   {
      /* append table heading */

      StringBuilder buffer = new StringBuilder(1000);
      buffer.append("  RATE");
      for (int years = minYears; years <= maxYears; years += yearStep)
      {
         String head = String.format("%2d YEARS", years);
         buffer.append(String.format("%12s", head));
      }
      buffer.append("\n");

      /* Calculate and append rows of table */

      double small = 0.00001;
      for (double rate = minRate; rate <= maxRate + small; rate += rateStep)
      {
         buffer.append(String.format("%6.2f", rate));
         for (int years = minYears; years <= maxYears; years += yearStep)
         {
            double value = futureValue(initialValue, rate, years);
            buffer.append(String.format("%12.2f", value));
         }
         buffer.append("\n");
      }
      table = buffer.toString();
   }

   private double futureValue(double presentValue, double yearlyRate, int years)
   {
      double monthlyRate = yearlyRate / 100.0 / 12.0;
      return presentValue * Math.pow(1.0 + monthlyRate, 12 * years);
   }
}
```

7.11.2 Console user interface

Here is a runner class that can be used in BlueJ and from the command line.

Class `InvestmentTableRunner`

book-projects/chapter7/investment

```
package chapter7.investment; // remove this line if you’re not using packages
import java.util.Scanner;
/**
```

```
 * Console runner class for InvestmentTable.
 * This version allows all table parameters to be specified.
 */
public class InvestmentTableRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);

      System.out.println("Enter minimum rate in percent");
      double minRate = input.nextDouble();
      input.nextLine();
      System.out.println("Enter table rate step");
      double rateStep = input.nextDouble();
      input.nextLine();
      System.out.println("Enter maximum rate in percent");
      double maxRate = input.nextDouble();
      input.nextLine();


      System.out.println("Enter minimum number of years");
      int minYears = input.nextInt();
      input.nextLine();
      System.out.println("Enter table year step");
      int yearStep = input.nextInt();
      input.nextLine();
      System.out.println("Enter maximum number of years");
      int maxYears = input.nextInt();
      input.nextLine();


      System.out.println("Initial investment");
      double amount = input.nextDouble();
      input.nextLine();
      InvestmentTable table = new InvestmentTable(minRate, rateStep, maxRate,
         minYears, yearStep, maxYears, amount);
      System.out.println(table);
   }

   public static void main(String[] args)
   {
      new InvestmentTableRunner().run();
   }
}
```

Typical console output is

```
java InvestmentTableRunner
Enter minimum rate in percent
2
Enter table rate step
0.5
Enter maximum rate in percent
```

```
5
Enter minimum number of years
1
Enter table year step
1
Enter maximum number of years
5
Initial investment
1000
  RATE     1 YEARS     2 YEARS     3 YEARS     4 YEARS     5 YEARS
  2.00     1020.18     1040.78     1061.78     1083.21     1105.08
  2.50     1025.29     1051.22     1077.80     1105.06     1133.00
  3.00     1030.42     1061.76     1094.05     1127.33     1161.62
  3.50     1035.57     1072.40     1110.54     1150.04     1190.94
  4.00     1040.74     1083.14     1127.27     1173.20     1221.00
  4.50     1045.94     1093.99     1144.25     1196.81     1251.80
  5.00     1051.16     1104.94     1161.47     1220.90     1283.36
```

7.12 Plotting the graph of a function

As another example of a for-loop let us write a graphics program to plot a function $f(x)$ for $x_L \leq x \leq x_R$. To draw the graph we have to approximate the curve by many line segments and draw each line segment. If we choose small enough line segments the graph will look smooth. Therefore, we divide the interval $[x_L, x_R]$ into n equal size subintervals using the points

$$x_0 = x_L,\, x_1 = x_L + dx,\, \ldots,\, x_i = x_L + i\,dx,\, \ldots,\, x_n = x_R = x_L + n\,dx,$$

where $dx = (x_R - x_L)/n$. The graph between x_i and x_{i+1} is shown in Figure 7.15. It shows the

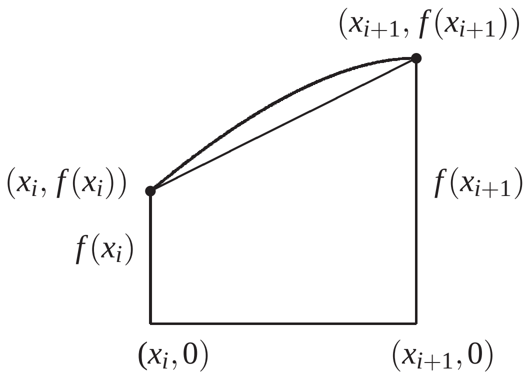


Figure 7.15: Approximating part of a function with a line segment

curve and the line segment that is used to approximate it. On this interval we need to draw a line from the point $(x_i,\, f(x_i))$ to the point $(x_{i+1},\, f(x_{i+1}))$. A pseudo-code algorithm for drawing the graph of $f(x)$ is shown in Figure 7.16. Here each time a line segment is drawn its right end point becomes the left end point of the next line segment. This is accomplished with the assignment $(x_0, y_0) \leftarrow (x, y)$.

```
**ALGORITHM** DrawGraph($x_L$, $x_R$, $n$)
$dx \leftarrow (x_R - x_L)/n$
$(x_0, y_0) \leftarrow (x_L, y_L)$
**FOR** $i \leftarrow 1$ **TO** $n$ **DO**
    $(x,y) \leftarrow (x_L, +i\,dx, f(x_L, +i\,dx))$
    Draw line from $(x_0, y_0)$ to $(x,y)$
    $(x_0, y_0) \leftarrow (x,y)$
**END FOR**
```

Figure 7.16: Pseudo-code graph drawing algorithm

As an example, let us write a class to draw the graph of $\sin x$ from $x_L = -2\pi$ to $x_R = 2\pi$. The natural world coordinate system is one that has this range on the x-axis, and the range -1 to 1 on the y-axis.

We can use the `GraphicsFrame` class and the `worldTransform` method from the `BarGraph3` class in Chapter 5 (page 239). The bounding box for the world coordinate system can be defined using

```
private double xLeft = -2 * Math.PI;
private double xRight =  2 * Math.PI;
private double yBottom = -1.0;
private double yTop = 1.0;
```

The `paintComponent` method has the basic structure

```
public void paintComponent(Graphics g)
{
   super.paintComponent(g);
   Graphics2D g2D = (Graphics2D) g;

   int w = getWidth();
   int h = getHeight();

   int numPoints = 100;
   double b = 1.1;
   AffineTransform world = worldTransform(xLeft, xRight, yBottom*b, yTop*b, w, h);
   g2D.transform(world);

   // choose a line thickness here
   // Draw the x and y axes here
   // Draw the graph of sin x here
}
```

Here `b` provides a 10% border in the y direction so the graph doesn't touch the edge of the window.

Recall that any transformation of user space involving a scaling also transforms the line thickness. We want our lines to be one pixel in size so we need to calculate the width and height of a pixel in the world system and use the smallest of them as our pixel size. Since `h` pixels vertically correspond to `yTop - yBottom` units in the world and `w` pixels horizontally correspond to `xRight - xLeft` units we obtain

```
double pixelHeight = (yTop - yBottom) / h;
double pixelWidth = (xRight - xLeft) / w;
double pixelSize = Math.min(pixelHeight, pixelWidth);
```

Now we can set the line size using

```
g2D.setStroke(new BasicStroke((float)pixelSize));
```

Alternatively, if we want a one pixel line we can use `0F` as the argument of `BasicStroke` (see Chapter 5).

The axes can be drawn in blue using

```
Line2D.Double xAxis = new Line2D.Double(xLeft,0,xRight,0);
Line2D.Double yAxis = new Line2D.Double(0,yBottom,0,yTop);
g2D.setPaint(Color.blue);
g2D.draw(xAxis);
g2D.draw(yAxis);
```

Finally, the graph can be drawn in black using `Point2D.Double` objects called `p0` and `p1` for the points (x_0, y_0) and (x, y) shown in the pseudo-code algorithm:

```
double dx = (xRight - xLeft) / numPoints;
Point2D.Double p0 = new Point2D.Double(xLeft, Math.sin(xLeft));
g2D.setPaint(Color.black);
for (int i = 1; i <= numPoints; i++)
{
   double x = xLeft + i*dx;
   double y = Math.sin(x);
   Point2D.Double p1 = new Point2D.Double(x,y);
   g2D.draw(new Line2D.Double(p0,p1));
   p0 = p1;
}
```

7.12.1 `SineGraph` class

Here is the complete class. The graph is shown in Figure 7.17.

Class `SineGraph`

book-projects/chapter7/sine_graph

```
package chapter7.sine_graph; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
```

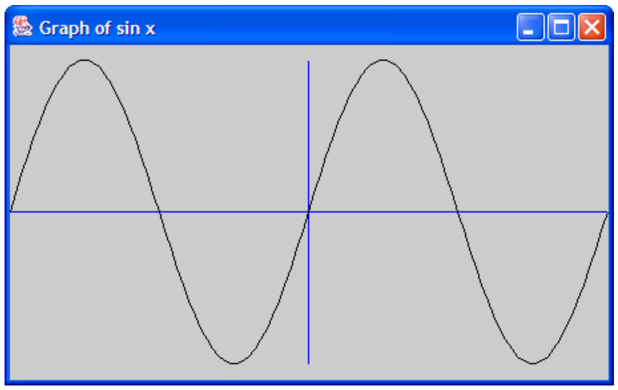


Figure 7.17: `SineGraph` output window

```
/**
 * Use the GraphicsFrame class to draw the graph
 * of a sine curve from -2*Math.PI to 2*Math.PI.
 * An affine transformation is used to transform the coordinate system.
 */
public class SineGraph extends JPanel
{
   // define bounding box for graph in world coordinate system

   private double xLeft = -2 * Math.PI;
   private double xRight =  2 * Math.PI;
   private double yBottom = -1.0;
   private double yTop = 1.0;

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      int w = getWidth();
      int h = getHeight();

      int numPoints = 100;

      // Make a world coordinate system to go from xLeft*b to xRight*b in the
      // x direction and yBottom*b to yTop*b in the y direction, where b is
      // chosen to leave a 10 percent border in y direction

      double b = 1.1;
      AffineTransform world = worldTransform(xLeft, xRight,
         yBottom*b, yTop*b, w, h);
      g2D.transform(world);

      // find out the size of a pixel and use it to scale the
      // line thickness to one pixel. pixelHeight and pixelWidth
      // are the pixel width and height in the world system.
      // Choose their minimum as the width of lines.

      double pixelHeight = (yTop - yBottom) / h;
      double pixelWidth = (xRight - xLeft) / w;
      double pixelSize = Math.min(pixelHeight, pixelWidth);
      g2D.setStroke(new BasicStroke((float)pixelSize));

      // Another way to get one pixel lines is to use 0F as the brush size

      // Draw the x and y axes

      Line2D.Double xAxis = new Line2D.Double(xLeft,0,xRight,0);
      Line2D.Double yAxis = new Line2D.Double(0,yBottom,0,yTop);
      g2D.setPaint(Color.blue);
      g2D.draw(xAxis);
      g2D.draw(yAxis);

      // Determine the distance on the x-axis between successive points

      double dx = (xRight - xLeft) / numPoints;

      // Set the starting point on the curve to the leftmost point.

      Point2D.Double p0 = new Point2D.Double(xLeft, Math.sin(xLeft));

      // Draw numPoints line segments. The x coordinates of their right end
      // points are xLeft+dx, xLeft+2*dx, ... and so on. After drawing a
      // segment reset the starting point p0 for the next line segment.

      g2D.setPaint(Color.black);
      for (int i = 1; i <= numPoints; i++)
      {
         double x = xLeft + i*dx;
         double y = Math.sin(x);
         Point2D.Double p1 = new Point2D.Double(x,y);
         g2D.draw(new Line2D.Double(p0,p1));
         p0 = p1;
      }
   }

   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h)
   {
      double sx = (w-1) / (xMax - xMin); // scale factor in x direction
      double sy = (h-1) / (yMax - yMin); // scale factor in y direction
      AffineTransform at = new AffineTransform();
      at.translate(0, h-1);  // move origin to bottom left corner of JPanel
      at.scale(sx, -sy);     // -sy reverses y axis
      at.translate(-xMin, -yMin);  //make (xMin,yMin) the lower left corner
      return at;
   }
   public void draw()
   {
      new GraphicsFrame("Graph of sin x", new SineGraph(), 301, 201);
   }

   public static void main(String[] args)
   {
      new SineGraph().draw();
   }
}
```

7.13 Recursion and loops

7.13.1 What is recursion?

Recursion is a problem solving technique that expresses a problem (algorithm) in terms of one or more smaller versions of itself. These smaller versions are, in turn, expressed in terms of smaller versions of themselves, and so on. The smaller versions of the problem at each stage are called the **recursive cases**. This process continues until we arrive at one or more cases which can be solved directly. These cases are called **base cases**.

7.13.2 Examples of recursive definitions

The simplest forms of recursion have a close connection with loops. Many functions in mathematics have non-recursive definitions expressed in terms of loops and recursive definitions that do not have explicit loops: the looping process is managed by the recursion process itself, as it breaks the problem into smaller subproblems.

We give three examples.

■ EXAMPLE 7.24 **(Recursive definition of $n!$)** In Section 7.8.1 we gave a non-recursive definition of $n!$. A recursive definition is

$$\begin{array}{ll} 0! = 1, \quad 1! = 1 & \text{(base cases, } n = 0, 1) \\ n! = n(n-1)! & \text{(recursive cases, } n > 1) \end{array}$$

The base cases are 0! and 1!. The recursive cases express $n!$ in terms of $(n-1)!$, a smaller version of itself. ■

■ EXAMPLE 7.25 **(Recursive definition of the Fibonacci numbers)** The Fibonacci numbers $F_n, n = 0, 1, \ldots$ are defined by

$$\begin{array}{ll} F_0 = 0, \quad F_1 = 1 & \text{(base cases, } n = 0, 1) \\ F_n = F_{n-1} + F_{n-2} & \text{(recursive cases, } n > 1) \end{array}$$

Here there are two simple base cases and the recursive cases express F_n as the sum of two smaller versions, F_{n-1} and F_{n-2} ■

■ $\boxed{\text{EXAMPLE 7.26}}$ **(Recursive definition of the greatest common divisor)** The greatest common divisor of two integers m and n is denoted by $\gcd(m,n)$. It is the largest integer that divides both m and n. We can assume that $m \geq 0$ and $n \geq 0$ since $\gcd(m,n) = \gcd(|m|,|n|)$. Under these assumptions a recursive definition for $\gcd(m,n)$ is

$$\begin{array}{ll} \gcd(m,0) = m & \text{(base cases, } n = 0\text{)} \\ \gcd(m,n) = \gcd(n, m \bmod n) & \text{(recursive cases, } n > 0\text{)} \end{array}$$

We could also assume that $m \geq n$ since $\gcd(m,n) = \gcd(n,m)$, although the recursive definition works in either case. The base cases, when $n = 0$, do not involve recursion. The recursive cases involve smaller versions of $\gcd(m,n)$ since $m \bmod n$ is smaller than n so the second argument, n, decreases until the base case is reached at $n = 0$. For example

$$\begin{aligned} \gcd(2436,1015) &= \gcd(1015, 2436 \bmod 1015) = \gcd(1015,406) \\ &= \gcd(406, 1015 \bmod 406) = \gcd(406,203) \\ &= \gcd(203, 406 \bmod 203) = \gcd(203,0) \\ &= 203 \end{aligned}$$

so $\gcd(2436,1015) = 203$. ■

7.13.3 Recursive factorial method

To see the connection with loops recall that the non-recursive version of the factorial function in program `FactorialCalculator` (page 338) involved a for-loop. Using the recursive definition in Example 7.24 the recursive version of this function is given in the following example.

■ $\boxed{\text{EXAMPLE 7.27}}$ **(Recursive method for $n!$)**

```
int factorial(int n)
{
   if (n == 0 || n == 1) // base cases
      return 1;
   else // recursive case
      return n * factorial(n-1);
}
```

Notice that the function calls itself but with a smaller version of the argument. Eventually the function will be called with 1 as a argument and the base case will stop the recursion. There is no loop in this version of the factorial function. The recursive process itself does the looping: each of the pending returns does one of the multiplications. Of course, if the base case is omitted then we have what is called infinite recursion, corresponding to an infinite loop. ■

The following class can be used to test the recursive factorial function.

Class `FactorialCalculator`

book-projects/chapter7/recursion

```
package chapter7.recursion; // remove this line if you’re not using packages
/**
 * A simple class to test recursive version of factorial method.
 */
public class FactorialCalculator
{
   /**
    * Calculate n factorial using recursive algorithm.
    * @param n value for n factorial
    * @return n factorial
    * @throws IllegalArgumentExeception if n is outside the range
    * 0 &lt;= n &lt;= 12.
    */
   public int factorial(int n)
   {
      if (n < 0 || n > 12)
      {
         throw new IllegalArgumentException("n! is only defined for n = 0..12");
      }

      if (n == 0 || n == 1) // base cases
         return 1;
      else // recursive cases
         return n * factorial(n-1);
   }
}
```

A runner class for testing the method from the command line is given by

Class `FactorialRunner`

book-projects/chapter7/recursion

```
package chapter7.recursion; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing recursive factorial method from commmand line
 */
public class FactorialRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      FactorialCalculator calc = new FactorialCalculator();
      System.out.println("Enter value of n");
      int n = input.nextInt();
      System.out.println(n + "! = " + calc.factorial(n));
   }
   public static void main(String[] args)
   {
      new FactorialRunner().run();
   }
}
```

To see how the recursive process works let us calculate 4!. The steps are

$$\begin{aligned} 4! &\rightarrow 4\cdot 3! \\ &\rightarrow 4\cdot 3\cdot 2! \\ &\rightarrow 4\cdot 3\cdot 2\cdot 1! \;\; \text{base case} \\ &\rightarrow 4\cdot 3\cdot 2\cdot 1 \\ &\rightarrow 4\cdot 3\cdot 2 \\ &\rightarrow 4\cdot 6 \\ &\rightarrow 24 \end{aligned}$$

The recursive calls end with the base case, then the pending returns do all the multiplications.

7.13.4 Recursive gcd method

The following example gives a method for the recursive gcd algorithm in Example 7.26.

■ EXAMPLE 7.28 **(Recursive method for** $\gcd(m,n)$**)**

```
     int gcd(int m, int n)
     {
        if (n == 0)
           return m;
        else
           return gcd(n, m % n);
        }
```

In case $m = n = 0$ the method returns 0 even though $\gcd(0,0)$ is not normally defined. Insert the statements

```
     m = Math.abs(m);
     n = Math.abs(n);
```

if you want the method to also work in case one or both of *m* and *n* are negative. ■

The following class can be used to test the recursive gcd function.

Class `GcdCalculator`

book-projects/chapter7/recursion

```
package chapter7.recursion; // remove this line if you’re not using packages
/**
 * A simple class to test recursive version of gcd method.
 */
public class GcdCalculator
{
   /**
    * Calculate gcd(m,n) using recursive algorithm.
    * m >= 0 and n >= 0. Algorithm produces gcd(0,0) = 0
    * even though gcd(0,0) is undefined.
    * @return gcd(m,n)
    */
   public int gcd(int m, int n)
   {
      if (n == 0)
         return m;
      else
         return gcd(n, m % n);
   }
}
```

A runner class for testing the method from the command line is given by

Class `GcdRunner`

book-projects/chapter7/recursion

```
package chapter7.recursion; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * Testing recursive gcd method from commmand line
 */
public class GcdRunner
{
   public void run()
   {
      Scanner input = new Scanner(System.in);
      GcdCalculator calc = new GcdCalculator();
      System.out.println("Enter value of m");
      int m = input.nextInt();
      input.nextLine();
      System.out.println("Enter value of n");
      int n = input.nextInt();
      input.nextLine();

      System.out.println("gcd(" + m + ", " + n + ") = " + calc.gcd(m,n));
   }

   public static void main(String[] args)
   {
      new GcdRunner().run();
   }
}
```

7.13.5 Non-recursive and recursive sum methods

As another example of the connection between loops and recursion consider the problem of writing a method to compute the sum

$$S(a,b) \quad = \quad a+(a+1)+\cdots+(b-1)+b$$

of the integers between a and b for $a \le b$. Pretending that we don't know the answer

$$S(a,b) \quad = \quad \frac{b(b+1)}{2} - \frac{(a-1)a}{2} = \frac{(b+a)(b-a+1)}{2}$$

the non-recursive solution is to use a for-loop as in the simple method

```
public int sum(int a, int b)
{
   int s = 0;
   for (int k = a; k <= b; k++)
   {
      s = s + k;
   }
   return s;
}
```

We can also give a recursive definition of this sum in terms of smaller sums:

> "*The sum of the numbers from a to b is the first number plus the sum of the remaining numbers.*"

The smaller version of the problem is "sum of the remaining numbers", since there is one less number in this sum, and the base case occurs when there is a single number to sum. Assuming that $a \le b$ we have the following recursive definition

$$\begin{array}{ll} S(a,a) = a & \text{(base cases, } b = a\text{)} \\ S(a,b) = a + S(a+1,b) & \text{(recursive cases, } b > a\text{)} \end{array}$$

Here is a recursive method to compute the sum:

```
public int sum(int a, int b)
{
   if (a == b) // base case
      return a;
   else
      return a + sum(a+1, b);
}
```

7.14 Common loop errors

There are several errors that commonly occur when writing loops.

7.14.1 Misplaced semi-colon

Referring to Example 7.1, consider the simple while-loop

```
int count = 1;
while (count <= 10);
{
   System.out.print(count + " ");
   count = count + 1; // or use count++
}
```

Instead of displaying `1 2 3 4 5 6 7 8 9 10` nothing is displayed and the program doesn't stop. There is a semi-colon at the end of the line containing `while`. This is not a syntax error! This line is a complete while-loop with no body. Since `count <= 10` is true it will always be true and the empty loop will never exit. The statements enclosed in braces are not part of the loop and will never be executed.

7.14.2 Off by one errors

It is important to test loops to make sure the loop variables begin and end with the required values. It is easy to be "off by one" and have a loop that executes one less time or one more time. These logical errors can be difficult to find.

7.15 BeanShell exercises

The following BeanShell exercises can be done using the Workspace Editor. First run BeanShell, then choose "Workspace Editor" from the "File" menu to open the editor. If you want to use `System.out.println` then it is also necessary to choose "Capture System in/out/err" from the "File" menu.

Now you can type statements into the editor and they won't be executed as they are entered. When you have finished entering statements choose "Evaluate in Workspace" from the "Evaluate" menu. Now the statements will be executed. You can edit the statements and evaluate them again, and so on.

This is useful for testing static methods. Type in the method, evaluate it then test it interactively using the workspace.

► **BeanShell Exercise 7.1** Write some statements to compute the sum $S_n = 1 + 2 + \cdots + n$ using

(a) a for-loop that counts up,

(b) a for-loop that counts down,

(c) a while-loop that counts up,

(d) a while-loop that counts down,

(e) a do-while loop that counts up,

(f) a do-while loop that counts down.

Test your statements using the BeanShell editor and workspace.

▶ **BeanShell Exercise 7.2** Repeat Exercise 7.1 for the sum $O_n = 1+3+5+\cdots+(2n-1)$ of the first n odd numbers.

▶ **BeanShell Exercise 7.3** Repeat Exercise 7.1 for the sum $E_n = 2+4+6+\cdots+2n$ of the first n even numbers.

▶ **BeanShell Exercise 7.4** Write a method with prototype

```
int power(int m, int k)
```

that uses a for-loop to compute m^k for $k \geq 0$. Test your method using the BeanShell editor and workspace.

▶ **BeanShell Exercise 7.5** Write a method with prototype

```
double power(double m, int k)
```

that uses a single for-loop to compute m^k where k is any integer (positive or negative). Hint: distinguish the cases $k < 0$ and $k \geq 0$, multiplying m by itself $k-1$ times if $k > 0$ but multiplying $1/m$ by itself $-k$ times if $k < 0$. Test your method using the BeanShell editor and workspace.

7.16 Programming exercises

▶ **Exercise 7.1 (Computing factorials with the long data type)**
Write a version of `FactorialCalculator` called `LongFactorialCalculator` that uses the `long` data type for calculating $n!$. Also write a runner class called `LongFactorialRunner`. What is the largest value of n that can be used before overflow occurs?

▶ **Exercise 7.2 (Number digits in $n!$)**
The number of digits d in $n!$ is $1+\lfloor p \rfloor$ where $n! = 10^p$. Taking logarithms to base 10 gives the formula

$$d = 1 + \lfloor \log_{10} 2 + \log_{10} 3 + \cdots + \log_{10} n \rfloor$$

Write a method called `factorialDigits` and a tester class for this formula and test it using `BigFactorialRunner`. In the `Math` class there is the function `floor` with prototype

```
public static long floor(double n)
```

and there is the logarithm function to base e with prototype

```
public static double log(double n)
```

To get logarithms to base 10 use the formula $\log_{10} n = \dfrac{log_e n}{log_e 10}$.

▶ **Exercise 7.3 (Trailing zeros in n!)**
It can be shown that the number of trailing zeros in $n!$ is

$$n \text{ div } 5 + n \text{ div } 5^2 + n \text{ div } 5^3 + \cdots + n \text{ div } 5^k + \cdots$$

Terms in the sum with $5^k > n$ do not contribute since $n \text{ div } 5^k$ is zero. Write a class called `TrailingZeros` that uses the `int` data type to compute this sum. Use a while-loop with condition $5^k \le n$ in a method with prototype

```
int zeros(int n)
```

that returns the number of zeros. Also use a method with prototype

```
public int power(int m, int k)
```

that computes m^k ($k > 0$) using a for-loop and can be used as `power(5,k)` by the `zeros` method. You can use `BigFactorialTester` to test this formula.

▶ **Exercise 7.4 (Powers of two)**
Write a class called `PowersOfTwoCalculator` that computes and displays, for a given value of n, the first n powers of 2, namely $2^1, 2^2, \ldots, 2^n$. Do not use the `Math.pow` function. The program output for $n = 4$ should look like

```
Enter the largest power
4
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
```

What is the largest power that you can compute as an `int`?

▶ **Exercise 7.5 (Powers of two using BigInteger objects)**
Try the previous exercise by writing a class called `BigPowersOfTwo` that uses `BigInteger` objects.

▶ **Exercise 7.6 (Computing integer powers)**
Write a class called `IntegerPowerCalculator` that tests the method in BeanShell Exercise 7.5 for computing integer powers. Also write a runner class `IntegerPowerRunner` that can be run from the command line.

▶ **Exercise 7.7 (Computing e using a series)**
The base of the natural logarithms has the representation

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{k!} + \cdots$$

as an infinite sum. The n-th partial sum of this series is defined as

$$S_n = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}, \quad n \ge 0$$

As n increases these partial sums get closer and closer to the value of e. The nth term in this series is $t_n = 1/n!$. Write a class called `ExpCalculator` that uses a for-loop to compute these sums for $0 \le n \le nMax$ Where the value of $nMax$ is input by the user. You do not need to compute any factorials. Instead use the fact that

$$\frac{t_k}{t_{k-1}} = \frac{(k-1)!}{k!} = \frac{(k-1)!}{k(k-1)!} = \frac{1}{k}$$

to write $t_k = t_{k-1}/k$, where $t_0 = 1$. This expresses each term in terms of the preceding one. Then the partial sum S_k can be expressed in terms of the preceding one using $S_k = S_{k-1} + t_k$ and $S_0 = 1$. Now you can use a loop based on the pseudo-code algorithm in Figure 7.18 which displays the

```
**ALGORITHM** ExpCalculator($nMax$)
$s \leftarrow 1.0$
**OUTPUT** 0, $s$
$t \leftarrow 1.0$
**FOR** $k \leftarrow 1$ **TO** $nMax$ **DO**
    $s \leftarrow s + t$
    $t \leftarrow t/k$
    **OUTPUT** $k$, $s$
**END FOR**
```

Figure 7.18: Partial sum algorithm for *e*

partial sum number (0, 1, 2, ..., *nMax*) and the partial sum. Here *s* denotes a partial sum and *t* denotes a term. Your program should also display the approximate value of *e* using the constant `Math.E`.

► Exercise 7.8 (Computing e^x using a series)
Adapt the pseudo-code algorithm of Exercise 7.7 to compute partial sums of the series for e^x given by

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$$

using a class called `ExpXCalculator`. The input is now x and *nMax*. Compare your results by displaying `Math.exp(x)` after the last partial sum is displayed. Consider cases such as $x = 0.01$, $x = 0.1$, $x = 1$, and $x = 10$ and determine how many terms in the sum are needed to get agreement with `Math.exp(x)`.

► Exercise 7.9 (Computing $\sin x$ and $\cos x$ using a series)
Adapt the pseudo-code algorithm of Exercise 7.7 to compute partial sums of the series for $\sin x$ and $\cos x$ given by

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots + \frac{(-1)^k x^{2k+1}}{(2k+1)!} + \cdots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + \frac{(-1)^k x^{2k}}{(2k)!} + \cdots$$

using classes called `SinXCalculator` and `CosXCalculator`. The input is now x and *nMax*. Compare your results by displaying `Math.sin(x)` and `Math.cos(x)` after the last partial sum is displayed. Consider cases such as $x = 0.01$, $x = 0.1$, $x = 1$, and $x = 10$ and determine how many terms in the sum are needed to get agreement with `Math.sin(x)` and `Math.cos(x)`.

► Exercise 7.10 (A triangle of asterisks)
Write a class called `TriangleRight` that inputs a value of n defining the number of rows in the triangle and displays output like

```
   *
  **
 ***
****
```

which is the case $n = 4$.

► Exercise 7.11 (Another triangle of asterisks)
Write a class called `TriangleCenter` that inputs a value of n defining the number of rows in the triangle and displays output like

```
   *
  ***
 *****
*******
```

which is the case $n = 4$.

► Exercise 7.12 (Reversing a string)
Write a program class called `ReverseString` that tests a method called `reverse` with prototype

```
public String reverse(String s)
```

The method returns a string that is the reverse of `s`. For example, if `s` is `"Help"` the string returned is `"pleH"`. Use a for-loop that implements the steps shown in the table

i	reverse
0	`"h"` ← `"h" + ""`
1	`"eh"` ← `"e" + "h"`
2	`"leh"` ← `"l" + "eh"`
3	`"pleh"` ← `"p" + "leh"`

in the case that `s` is `"Help"`.

► **Exercise 7.13 (Nested while-loops)**
Write a complete class called `MarksCalculator` that reads a series of integer marks for a number of students using a negative mark to signal the end of the marks for each student. The program then calculates and prints the average of the marks for each student as a `double` number. After calculating and printing each average, the program asks if the user wants to enter marks for another student. A reply of “N” or “n” terminates the program. For any other reply the program continues with the next student. Use the following loop structure:

```
while (moreStudents)
{
   ...
   while (mark >= 0)
   {
      ...
   }
   ...
}
```

Some typical program output is

```
Enter marks for a student terminated by a negative mark
65
85
90
-3
The average for this student is 80
Do you want to enter marks for another student [Y/N]?
Y
Enter marks for a student terminated by a negative mark
55
75
70
-2
The average for this student is 66.67
Do you want to enter marks for another student [Y/N]?
N
```

► **Exercise 7.14 (Recursive calculation of Fibonacci numbers)**
The Fibonacci numbers were defined recursively in Example 7.25. Write a method with prototype

```
public int fibonacci(int n)
```

to compute Fibonacci numbers using this recursive definition. Put your method in a class called `RFibonacciCalculator` to test the method.

► **Exercise 7.15 (Non-recursive calculation of Fibonacci numbers)**
Do the previous exercise using a non-recursive method. Hint: each Fibonacci number is just the sum of the previous two numbers so keep track of two successive numbers so you can calculate the next one.

► Exercise 7.16 (Recursive binary gcd algorithm)
Here is an interesting recursive definition of $\gcd(m,n)$ for $m \geq n \geq 0$ that uses only simple subtraction and division by 2 operations.

(a) If $m < n$ then swap m and n.

(b) If $m = 0$ then $\gcd(m,n) = n$ (base case).

(c) If $n = 0$ then $\gcd(m,n) = m$ (base case).

(d) If m and n are both even then $\gcd(m,n) = 2\gcd(m/2,n/2)$.

(e) If m is odd and n is even then $\gcd(m,n) = \gcd(m,n/2)$.

(f) If m is even and n is odd then $\gcd(m,n) = \gcd(m/2,n)$.

(g) If m and n are both odd then $\gcd(m,n) = \gcd((m-n)/2,n)$.

Repeat the previous exercise using this definition and write a class called `BinaryGcdCalculator` to test your method.

► Exercise 7.17 (A non-recursive gcd algorithm)
Using the non-recursive pseudo-code algorithm for calculating $\gcd(m,n)$ shown in Figure 7.19, write a method with prototype

```
ALGORITHM gcd(m, n)
a ← m
b ← n
r ← a mod b
WHILE r ≠ 0 DO
    a ← b
    b ← r
    r ← a mod b
END WHILE
RETURN b
```

Figure 7.19: Pseudo-code non-recursive gcd algorithm

```
public int gcd(int m, int n)
```

Write a program called `GcdTester`, similar to `FactorialTester`, to test the method.

► Exercise 7.18 (A non-recursive gcd algorithm using subtraction)
Another interesting non-recursive pseudo-code algorithm for calculating $\gcd(m,n)$ that only uses subtraction is shown in Figure 7.20, where $m > 0$ and $n > 0$. Write a method with prototype

```
public int gcd(int m, int n)
```

```
ALGORITHM gcd($m,n$)
WHILE $m \neq n$ DO
    IF $m > n$ THEN
        $m \leftarrow m - n$
    ELSE
        $n \leftarrow n - m$
    END IF
END WHILE
RETURN $m$
```

Figure 7.20: A pseudo-code gcd algorithm using subtraction

that uses this algorithm. Write a class called `GcdSubtractTester`, similar to `GcdTester` in the preceding exercise to test the method.

► Exercise 7.19 (Ackermann's function)
Ackermann's function $A(m,n)$, for integers $m,n \geq 0$, has the recursive definition

$$A(m,n) = \begin{cases} n+1, \text{ if } m = 0 \\ A(m-1,1), \text{ if } n = 0,\ m > 0 \\ A(m-1,A(m,n-1)), \text{ if } m,n > 0 \end{cases}$$

Write a method with prototype

```
public int Ackermann(int m, int n)
```

for this function. Use a class called `AckermannCalculator`, similar to the `FactorialCalculator` class, so that you can test it. Evaluate $A(2,5)$ and $A(3,3)$. What happens when you try to evaluate $A(4,4)$?

► Exercise 7.20 (Pythagorean triples)
Pythagorean triples have the form (x,y,z) where x, y, and z are positive integers satisfying $x^2+y^2 = z^2$. They correspond to right angled triangles with integer sides x and y, containing the right angle, and hypotenuse z. Write a class called `PythagoreanTriples` that displays all triples having $x \leq 100$, $y \leq 100$, and $z \leq 100$. Generate only the triples in ordered form such that $x < y < z$. For example, $(3,4,5)$ will be generated but $(4,3,5)$ will not be generated.

► Exercise 7.21 (Unique Pythagorean triples)
In the previous exercise the triples (3,4,5), (6,8,10), and (9,12,15) are generated. Dividing the numbers in the second triple by 2 gives the first triple and dividing the numbers in the third triple by 3 also gives the first triple. For the triple (3,4,5) we have gcd(3,4) = 1, for the second triple gcd(6,8)= 2, and for the third triple gcd(9,12) = 3. There are many other cases like this.

Rewrite the program of the preceding exercise to generate only the triples (x,y,z) such that $\gcd(x,y) = 1$. You can use one of the gcd methods from the preceding exercises.

▶ **Exercise 7.22 (Generating prime numbers)**
A prime number is an integer $n \geq 2$ having no factors other than itself and 1. The first five prime numbers are $2, 3, 5, 7, 11$. Write a class called `PrimeGenerator` that takes two numbers *nMin* and *nMax* as input and displays all prime numbers p such that $nMin \leq p \leq nMax$.

▶ **Exercise 7.23 (Drawing a grid of lines)**
Write a graphics program called `GridMaker` that draws horizontal and vertical lines to form a 10 by 10 array of cells (as in a spreadsheet program). If the output window is resized the grid should expand or contract to fill it. An example is show in Figure 7.21. Here the cells are square but in

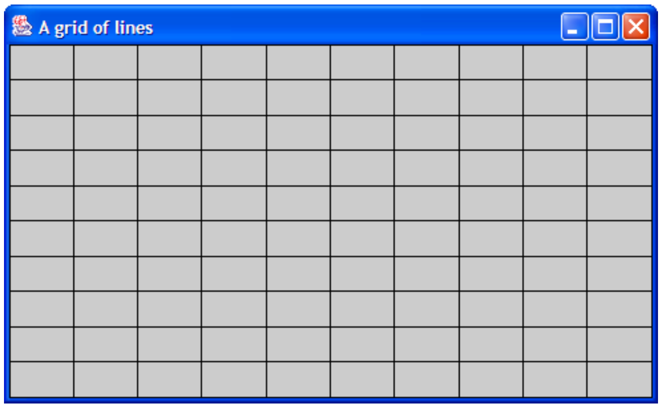


Figure 7.21: Output from the `GridMaker` class

general they will be rectangles.

▶ **Exercise 7.24 (Drawing concentric circles)**
Write a graphics program called `ConcentricCircles` that draws concentric circles in the output window, each with a different color. Use console input to get the number of circles from the user. The largest circle should just fit the window even if the window is resized. Color the circles (draw them from largest to smallest) using random colors obtained using the method

```
public Color randomColor()
{
   float red = (float) Math.random();
   float green = (float) Math.random();
   float blue = (float) Math.random();
   return new Color(red, green, blue);
}
```

which returns a random `Color` object for `setPaint`.

You may find the transformation

```
AffineTransform at = new AffineTransform();
at.translate(xMax/2.0, yMax/2.0);
g2D.transform(at);
```

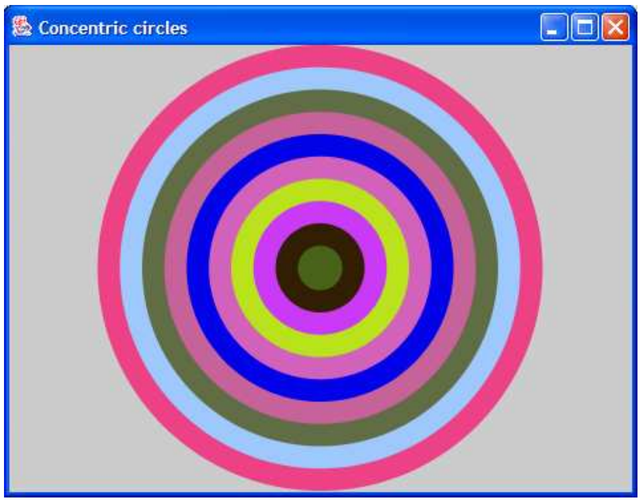


Figure 7.22: `ConcentricCircles` output window

useful for moving the coordinate origin to the center of the window. A sample output window is shown in Figure 7.22.

▶ **Exercise 7.25 (Drawing a regular polygon)**
The pentagon and hexagon were considered in Chapter 5. Write a `RegularPolygon` class that draws a regular n-sided polygon (all sides equal). Use console input to get the value of n from the user. The polygon should appear centered in the window. Fill the polygon with yellow and draw it with a brush 2 pixels wide. Hint: Assuming a center at $(0,0)$, the n vertices of the polygon are (x_k, y_k), $k = 0, \ldots n-1$, where $x_k = r\cos ka$, $y_k = r\sin ka$, r is the radius and the angle a is given by $a = 2\pi/n$ radians. If n is large enough the polygon will look like a circle.

▶ **Exercise 7.26 (Drawing a clock face)**
Write a graphics program called `ClockFace` that draws a circular clock face. The circle should be the largest one that fits the window. Draw tick marks every minute and double length tick marks every five minutes.

▶ **Exercise 7.27 (Drawing a ruler)**
Write a graphics program called `RulerMaker` that draws a picture of a ruler 10 cm wide, with centimeter marks in red, shorter 0.5 centimeter marks in blue and shorter millimeter marks in black.

▶ **Exercise 7.28 (Graphing functions)**
Using `SineGraph` as a model write a class called `SineCosGraph` that draws $\sin x$ and $\cos x$ on the same graph using different colors.

▶ **Exercise 7.29 (User interface for the PSRGame)**
In Chapter 6 the `PSRGameRunner` class was a console user interface for one round of the game. Write a better user interface that uses a query controlled while loop to play rounds of the game

until the users decide to quit. When the game ends display how many wins there were for each player.

Chapter 8

Array Data Types

Processing collections of objects

Outline

Mathematical sequences and subscript notation

Declaring and constructing arrays of primitive type

Sequential array processing

Declaring and constructing arrays of object type

String arrays and command-line arguments

Loan repayment and investment table examples

Arrays as method arguments and return values

Maximum array element algorithm

Linear search algorithm

Bubble sort algorithm

Efficient evaluation of a polynomial

A line graph using arrays

Two-dimensional arrays

Card deck application

8.1 Introduction

In this chapter we introduce the array data types to organize a sequence of data items so that they can be accessed using an index. The concept of an array is related to the subscript notation used for sequences in mathematics. Java arrays are objects so we first learn how to construct arrays of primitive types such as `int` and `double` and then we introduce arrays of object types (reference types). Arrays are designed to be processed using loops, especially the for-loop, so we discuss some array processing models and applications.

Some standard array processing algorithms are introduced: finding the maximum and minimum values in an array, searching for an element in an array using the linear search algorithm, sorting array elements in increasing order using the bubble sort algorithm, and the efficient evaluation of a polynomial.

Arrays of points or lines are also useful in graphics programs and as an example we write a class to draw a line graph given an array of points.

One-dimensional arrays can be generalized to *n*-dimensions and the important case of two-dimensional arrays is discussed using a matrix as an example.

The main problem with arrays in Java is that they are not dynamic. This means that the size of an array cannot be changed after it has been created so it is necessary to either know the size at compile-time or run-time. In later Chapters we will see how to use dynamic data types such as `ArrayList`.

8.2 Mathematical sequences and subscript notation

In programming languages the concept of an array derives from the similar concept of a subscripted variable in mathematics. A subscripted variable is useful when you want to define an ordered sequence of numbers or variables. For example, if $x_1, x_2, \ldots, x_n$ are n real numbers or variables then $\langle x_1, x_2, \ldots, x_n \rangle$ denotes the ordered sequence formed from them. Each subscript j with $1 \le j \le n$ is called an index. Sequences and subscripted variables are quite useful, as the following examples show.

■ $\boxed{\text{EXAMPLE 8.1}}$ **(Summation notation for sequences)** If $\langle x_1, x_2, \ldots, x_n \rangle$ is a sequence of real numbers then their average is defined by

$$\frac{x_1 + x_2 + \cdots + x_n}{n} = \frac{1}{n} \sum_{k=1}^{n} x_k$$

where the summation symbol denotes the sum of the variables x_k in the sequence. At the bottom of the summation symbol we put the index name and its initial value and at the top we put the final value of the index. Then x_k is a typical term in the sum. ■

■ $\boxed{\text{EXAMPLE 8.2}}$ **(Infinite sequences)** Sequences can be infinite. In this case a rule defines each value in the sequence. The geometric sequence $\langle a_0, a_1, \ldots, a_n, \ldots \rangle$ is defined, for some number r called the geometric ratio, by $a_k = r a_{k-1}$, for all integers $k \ge 1$. This defines each sequence value in terms of the previous one. $\langle 2, 6, 18, 54, \ldots \rangle$ is an example with $a_0 = 2$ and $r = 3$. ■

■ $\boxed{\text{EXAMPLE 8.3}}$ **(Sequences of vertices)** If the points $v_k = (x_k, y_k)$, $k = 0, \ldots, n-1$, are vertices of an n-sided polygon then the polygon can be defined as the vertex sequence $\langle v_0, v_1, \ldots, v_{n-1} \rangle$. ■

■ $\boxed{\text{EXAMPLE 8.4}}$ **(Matrix multiplication)** Consider the 3×3 matrices

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}, \quad C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}.$$

If C is the product of A and B, denoted by $C = AB$, then the matrix element c_{ij} in row i and column j of C is defined by

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} = \sum_{k=1}^{3} a_{ik}b_{kj}, \quad i = 1,2,3, \; j = 1,2,3$$

in terms of the matrix elements of A and B. Here we have a double subscript notation with the first subscript identifying the row of the matrix and the second subscript identifying the column.

This rule for matrix multiplication can be generalized. Suppose that A is an $m \times p$ matrix (m rows, p columns), and B is a $p \times n$ matrix. Then

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ik}b_{kj}, \quad 1 \le i \le m, \; 1 \le j \le n,$$

are the matrix elements of the $m \times n$ product matrix $C = AB$. ■

A finite sequence is represented in Java by an array. The mathematical subscript notation a_k is denoted using square brackets by `a[k]`. Sequences like this are called one-dimensional arrays. Matrices are examples of two-dimensional arrays and the mathematical notation a_{ij} is denoted using two sets of square brackets by `a[i][j]`. Array indices in Java always begin at the value 0, so if `a` is the name of an array with n elements then these elements are denoted by `a[0]`, `a[1]`, ..., `a[n-1]`.

8.3 Declaring and constructing arrays

We have seen that there are two kinds of variables: variables of primitive type and variables of object type (reference type). The same is true for arrays. There are arrays of primitive type and arrays of object type. In either case arrays are objects but the elements can be of a primitive type such as `int` or `double` or references to objects such as `BankAccount` objects.

8.3.1 Arrays of primitive type

Declaring array types

Corresponding to each primitive type there is an associated array type obtained by putting `[]` after the primitive type name. This gives the array types `int[]`, `double[]`, `char[]`, and `boolean[]` for an array of `int`, `double`, `char`, and `boolean` values, respectively, and similarly for the other primitive types. For example, the statement

```
int[] score;
```

declares that the variable `score` is the name of an array of integers. The type of this array is `int[]` so `score` is a reference to an array of integers. We say that `score` is an **array reference**.

Constructing arrays

Once an array reference has been defined we can construct an array for it to reference. Since arrays are objects this is done using `new`. For example we can associate a 5 element array with `score` using the statement

```
score = new int[5];
```

The keyword `new` is followed by the type of the array elements and then brackets containing the number of array elements. A common mistake is to assume that the 5 represents the highest index. Since indices begin at 0, the highest index is always one smaller than the value specified after `new`. Therefore this statement allocates storage space for 5 integer variables denoted by

```
score[0], score[1], score[2], score[3], score[4]
```

It is possible to define both the array reference and the array in one statement using

```
int[] score = new int[5];
```

Assigning values to array elements

Once an array has been constructed values can be assigned to the elements using assignment statements or array initializers.

■ EXAMPLE 8.5 **(Array assignment)** The statements

```
int[] score = new int[5];
score[0] = 1000;
score[1] = 3250;
score[2] = 2104;
score[3] = 675;
score[4] = 1454;
```

declare an array called `score`, construct the array, and assign values to the array elements using assignment statements. ■

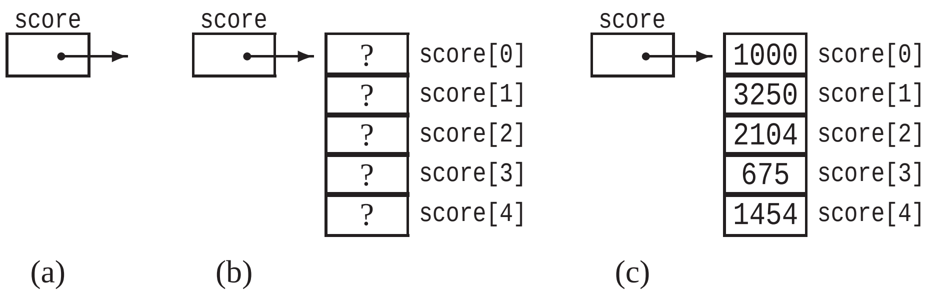


Figure 8.1: Constructing an array of integers

A pictorial representation of this three-step process is shown in Figure 8.1 for the `score` array. Part (a) shows the array reference before the array has been created, part (b) shows the situation after the array has been created. The question marks indicate that at this stage no values have been assigned to the elements. Finally, part (c) shows the array after the five assignment statements in Example 8.5 have been executed.

Using array initializers

Assignment statements give values to array elements at run-time. If you know what the values of the array elements are at compile-time (when you write the class), or if you have an array of constants, then it is more convenient to use an **array initializer** to assign values. An array initializer is a comma-separated list of values enclosed in braces. It can be used as the right side of an array declaration statement.

In the following three examples `new` is not used and the size of the array is not specified because the compiler automatically constructs the array and determines the appropriate size using the list of values in the initializer.

■ EXAMPLE 8.6 **(Array initializers)** The following statement uses an array initializer

```
int[] factorial =
   {1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600};
```

to define and construct an array for the factorials of the numbers 0 to 12. It is shorthand accepted by the compiler for

```
int[] factorial = new int[]
   {1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600};
```

Thus, 0! is `factorial[0]` which is 1, and 12! is `factorial[12]` which is `479001600`. In this example it would be better to use the `final` modifier to define the local array of constants

```
final int[] FACTORIAL =
   {1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600};
```

This informs the compiler that the values cannot be changed. We can also use

```
private static final int[] FACTORIAL =
   {1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600};
```

which makes the constant array a static data field. ■

■ EXAMPLE 8.7 **(Array initializers)** If you need the number of days in each month the array

```
final int[] DAYS_IN_MONTH = {31,28,31,30,31,30,31,31,30,31,30,31};
```

is useful. A leap year test can be used to obtain the correct value for February. ■

■ EXAMPLE 8.8 **(Array initializers)** If you need the day of the year, for a given month and day of the month, the array

```
final int[] DAY_NUMBER = {0,31,59,90,120,151,181,212,243,273,304,334};
```

is useful. Each entry is the number of days preceding the first day of each month (0 days precede January 1, 31 days precede February 1, etc.). For a leap year 1 can be added for months beyond February. ■

8.3.2 Calculating the number of days in a month

As a simple example let us write a class to solve the following problem.

"Given the year and the month what is the number of days in the month."

The only problem here is to account for the fact that February has 29 days in a leap year and 28 days otherwise. We have already written an `isLeapYear` method in Chapter 6, Example 6.30. The array in Example 8.7 can be used. This gives the statements

```
int days = DAYS_IN_MONTH[month-1];
if (isLeapYear(year) && month == 2) days++;
```

to calculate `days`, the number of days in the month.

Notice that `month` has a value in the range 1 to 12 but array indices always begin at zero so we needed to use `month-1` as the array index. This is an excellent example showing how arrays can be used to "look up" values. Here is a complete class to test the calculations.

Class `DaysInMonthCalculator`

book-projects/chapter8/simple_arrays

```
package chapter8.simple_arrays; // remove this line if you're not using packages
/**
 * Compute the number of days in a month given the year and the month.
 */
public class DaysInMonthCalculator
```

```
{
   private static int[] DAYS_IN_MONTH = {31,28,31,30,31,30,31,31,30,31,30,31};

   /**
    * Calculate number of days in a month
    * @param year the year to use
    * @param month the month in the range 1 to 12
    */
   public int daysInMonth(int year, int month)
   {
      int days = DAYS_IN_MONTH[month-1];
      if (isLeapYear(year) && month == 2) days++;
      return days;
   }

   /**
    * Return true of given year is a leap year else false.
    * @param year the year to test
    * @return true if year is a leap year else false
    */
   public boolean isLeapYear(int year)
   {
      return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0);
   }
}
```

We have used the `static` modifier in the definition of the array because it is not associated with any object. A suitable runner class that can be used from the console is given by

Class `DaysInMonthRunner`

book-projects/chapter8/simple_arrays

```
package chapter8.simple_arrays; // remove this line if you’re not using packages
import java.util.Scanner;
/**
 * A runner class for DaysInMonthCalculator
 */
public class DaysInMonthRunner
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter year");
      int year = input.nextInt();
      input.nextLine();
      System.out.println("Enter month (1 to 12)");
      int month = input.nextInt();
      input.nextLine();

      DaysInMonthCalculator calculator = new DaysInMonthCalculator();
      if (calculator.isLeapYear(year))
         System.out.println(year + " is a leap year");

      System.out.println("Number of days in month is "
         + calculator.daysInMonth(year, month));
   }
}
```

Declaring the size of an array at run-time

So far the size of our arrays has been declared at compile time. For example, the `score` array has size 5, the `FACTORIAL` array has size 13, and the `DAYS_IN_MONTH` array has size 12. The following example shows how to to define the size at run-time.

■ EXAMPLE 8.9 **(Run-time array size)** Assuming that `input` is a `Scanner` object the statements

```
System.out.println("Enter number of vertices in polygon");
int size = input.nextInt();
int[] v = new int[size];
```

declare and construct an array `v` to hold `size` elements where the value of `size` isn't known until the program is running. ■

Once the size of an array has been specified, either at compile-time or run-time, it cannot be changed. In this sense arrays in Java are not dynamic. However, there are situations where it is useful to have **dynamic arrays** that can be re-dimensioned at any time (increased or decreased in size). In later Chapters we will introduce classes that are dynamic.

The length of an array

The length, or size, of an array is the number of elements in the array. If `score` is an array then the length can be determined using the special notation `score.length`. This looks like a method call but there are no parentheses. You can think of `length` as a public instance data field for an array. Then `score.length` is an example of a qualified name for this field. It is a common mistake to use `score.length()` since this is the correct syntax for `String` objects.

8.3.3 Sequential array processing

Arrays can easily be processed sequentially using a for-loop. In fact, this is the primary use of a for-loop. For example, if `a` is an array, the standard loop structure

```
for (int k = 0; k < a.length; k++)
{
   // process array element a[k] here
}
```

can be used to access the array elements `a[k]` one element at a time, starting with `a[0]` and ending with `a[a.length-1]`. It is an error to use an array index that is out of bounds (negative or greater than `a.length - 1`). The Java interpreter will throw an `ArrayIndexOutOfBoundsException` for an invalid array index.

■ EXAMPLE 8.10 **(Sum and average of the elements of an array)** Assuming that the `score` array has already been defined as an array of `double` numbers, the statements

```
double sum = 0.0;
for (int k = 0; k < score.length; k++)
{
   sum = sum + score[k];
}
double average = sum / (double) score.length;
```

calculate the sum and average of its elements. ■

■ EXAMPLE 8.11 **(Displaying an array)** The for-loop

```
for (int k = 0; k < FACTORIAL.length; k++)
{
   System.out.println(k + "! = " + FACTORIAL[k]);
}
```

displays the factorials using the array declared in Example 8.6. ■

■ EXAMPLE 8.12 **(Reading arrays interactively)** If `input` is a `Scanner` object, statements similar to

```
System.out.println("Enter the number of array elements");
int size = input.nextInt();
input.nextLine();
double[] score = new double[size];
for (int k = 0; k < score.length; k++)
{
   System.out.println("Enter element " + k);
   score[k] = input.nextDouble();
   input.nextLine();
}
```

can be used to read the number of array elements, construct an array of this size, and use a for-loop to read values for the array elements. ■

8.3.4 Arrays of object type

So far we have considered only arrays whose elements are of a primitive type such as `int` or `double`. It is also common to define arrays of object type. Each array element contains a reference

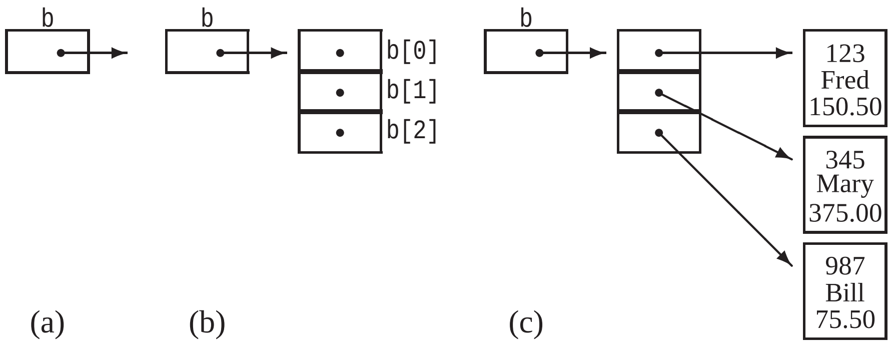


Figure 8.2: Constructing an array, `b`, of three `BankAccount` objects

to an object of a specified type. Thus the array is not really an array of objects, although this terminology is common, it is an array of references to objects. Constructing such an array is a three step process:

1. Declare an array reference variable.
2. Construct an array of references.
3. Construct some objects and assign their references to the array elements.

As for arrays of primitive type, steps 1 and 2 can be done separately, or they can be done together.

■ EXAMPLE 8.13 **(`BankAccount` array)** The statements

```
BankAccount[] b;
b = new BankAccount[3];
b[0] = new BankAccount(123, "Fred", 150.50);
b[1] = new BankAccount(345, "Mary", 375.00);
b[2] = new BankAccount(987, "Bill", 75.50);
```

illustrate these three steps. The first statement defines `b` as a reference to an array whose elements will be `BankAccount` object references. Next an assignment statement constructs the array of three references and the last three assignment statements construct `BankAccount` objects and associate them with the array of references. The `new` keyword is used in two ways here: first to construct the array of references, and then to construct objects for the array elements to reference. The single statement

```
BankAccount[] b = new BankAccount[3];
```

can also be used to declare the array reference and construct the array. ■

The entire process is shown in Figure 8.2 for this example. Part (a) corresponds to

```
BankAccount[] b;
```

and part (b) corresponds to

```
 b = new BankAccount[3];
```

At this stage we have the array of references indicated by the absence of arrows in part (b). Finally the assignment statements that construct three `BankAccount` objects result in the picture in part (c), showing three arrows referencing the three `BankAccount` objects.

■ EXAMPLE 8.14 **(Array initializers for object types)** The statements

```
BankAccount[] b = { new BankAccount(123, "Fred", 150.50),
   new BankAccount(345, "Mary", 375.00),
   new BankAccount(987, "Bill", 75.50) };
```

use an array initializer. The result is the same as in Example 8.13. ■

■ EXAMPLE 8.15 **(Totaling bank balances)** If `b` is an array of `BankAccount` objects, as in the preceding example, the statements

```
double totalBalance = 0.0;
for (int k = 0; k < b.length; k++)
{
   totalBalance = totalBalance + b[k].getBalance();
}
double averageBalance = totalBalance / b.length;
```

calculate the total of all balances and the average balance for accounts in the array. ■

`Point2D` arrays

In graphics programming it is common to use objects of type `Point2D.Double` and the following example shows how to define an array of points.

■ EXAMPLE 8.16 **(Array of Point2D objects)** The statements

```
Point2D.Double[] p = new Point2D.Double[3];
p[0] = new Point2D.Double(0,0);
p[1] = new Point2D.Double(1,2);
p[2] = new Point2D.Double(2,4);
```

define an array for three `Point2D.Double` objects corresponding to the points $(0,0)$, $(1,2)$, and $(2,4)$. The equivalent statement

```
Point2D.Double[] p = { new Point2D.Double(0,0), new Point2D.Double(1,2),
   new Point2D.Double(2,4) };
```

uses an array initializer. ■

■ EXAMPLE 8.17 **(Connecting an array of points with lines)** Given the array `p` of points in Example 8.16 and a graphics context `g2D`, the for-loop

```
for (int k = 0; k < p.length - 1; k++)
{
   g2D.draw(new Line2D.Double(p[k], p[k+1]));
}
```

connects the points with lines. Note that the largest k value is `p.length - 2` since n points define $n-1$ line segments. ■

8.3.5 String arrays

String arrays are common. For example, suppose we want to convert a month number in the range 1 to 12 to a name such as `"January"` or `"February"`. Without arrays this can be done with a large multiple if-statement. However, we can define a string array containing the month names and use the month number as an array index to "look-up" the name.

■ EXAMPLE 8.18 **(Array of month names)** The statement

```
final String[] MONTH_NAMES = { "January", "February", "March", "April",
   "May", "June", "July", "August", "September", "October",
   "November", "December" };
```

defines an array for the names of the 12 months. Now if `month` is a month number in the range 1 to 12, `MONTH_NAMES[month-1]` is the name of the month. ■

■ EXAMPLE 8.19 **(Array of day names)** Similarly, the statement

```
final String[] DAY_NAMES = { "Sunday", "Monday", "Tuesday", "Wednesday",
   "Thursday", "Friday", "Saturday" };
```

defines an array of strings for the names of the days of the week. ■

Command-line arguments

We can now explain the formal argument of the `main` method of a runner class, which we have not needed so far: it is an array of `String` objects. This array is used to store command-line arguments. When we write a main method such as

```
public static void main(String[] args)
{
   // statements
}
```

we are specifying that `args` is the name of an array of strings. These strings are the command-line arguments. When you use the Java interpreter to run a class the command-line arguments are typed after the name of the class on the command line. Each argument is separated by one or more spaces. For example, consider the following short class:

Class `CommandLineArguments`

book-projects/chapter8/simple_arrays

```
package chapter8.simple_arrays; // remove this line if you’re not using packages
/**
 * To show how command line arguments can be read int a String array that
 * is available to the program. The command line arguments are available
 * in the args array: args[0], args[1], ..., args[args.length - 1].
 */
public class CommandLineArguments
{
   public static void main(String[] args)
   {
      for (int k = 0; k < args.length; k++)
      {
         System.out.println("Argument " + k + " is " + args[k]);
      }
   }
}
```

When you run this class from the command line the first command line argument after the class name will be `args[0]`, the second will be `args[1]`, and so on. The number of command-line arguments typed is given by `args.length`. The `println` statement simply displays these arguments. Here is some sample output for two program runs:

```
java CommandLineArguments zero one two
Argument 0 is zero
Argument 1 is one
Argument 2 is two
java CommandLineArguments "zero one two"
Argument 0 is zero one two
```

The second example shows that spaces can be included in command-line arguments if the argument is enclosed in double quotes.

Command-line arguments can be quite useful for console programs that require only a few input items. For example, we can rewrite the `LoanRepaymentTableRunner` class from Chapter 7, page 348, so that it gets its four input values from the command line. Here is the new version of the class.

Class `LoanRepaymentTableRunner`

book-projects/chapter8/loan_repayment

```
package chapter8.loan_repayment; // remove this line if you’re not using packages
/**
 * Class for running LoanRepaymentTable from console using command-line args
 */
public class LoanRepaymentTableRunner
{
   public static void main(String[] args)
   {
      if (args.length == 4)
      {
         double a = Double.parseDouble(args[0]); // loan amount
         int y = Integer.parseInt(args[1]); // number of years
         int p = Integer.parseInt(args[2]); // payments per year
         double r = Double.parseDouble(args[3]); // annual rate in percent

         LoanRepaymentTable table = new LoanRepaymentTable(a,y,p,r);
         System.out.println(table);
      }
      else
      {
         System.out.println(
            "Args: amount years paymentsPerYear annualRate(percent)");
      }
   }
}
```

The `main` method checks to see if four command-line arguments were supplied, otherwise a message is displayed indicating the format of the arguments. Therefore if you forget the meaning of the arguments just run the class without any arguments. If four arguments are supplied the command-line strings are converted to the four arguments needed by the constructor using the static `Integer.parseInt` and `Double.parseDouble` methods in the wrapper classes introduced in Section 7.2.1. The output shown in Chapter 7 can be produced using the command

```
java LoanRepaymentTableRunner 10000 5 2 10
```

As another example we can rewrite `InvestmentTableRunner` from Chapter 7 (page 354), to use command-line arguments to obtain the seven constructor arguments. Here is the new version of the class.

Class `InvestmentTableRunner`

book-projects/chapter8/investment

```
package chapter8.investment; // remove this line if you're not using packages
import chapter7.investment.InvestmentTable; // remove this line if you're not using packages
/**
 * Class for running InvestmentTable from console using command-line args
 */
public class InvestmentTableRunner
{
   public static void main(String[] args)
   {
      if (args.length == 7)
      {
         double minRate = Double.parseDouble(args[0]);
         double rateStep = Double.parseDouble(args[1]);
         double maxRate = Double.parseDouble(args[2]);
         int minYears = Integer.parseInt(args[3]);
         int yearStep = Integer.parseInt(args[4]);
         int maxYears = Integer.parseInt(args[5]);

         double amount = Double.parseDouble(args[6]);
         InvestmentTable table =
            new InvestmentTable(minRate, rateStep, maxRate,
                                minYears, yearStep, maxYears, amount);
         System.out.println(table);
      }
      else
      {
        System.out.println(
           "Args: minRate rateStep maxRate minYears yearStep maxYears mount");
      }
   }
}
```

The output shown in Chapter 7 can be produced using the command

```
java InvestmentTableRunner 2 0.5 5 1 1 5 1000
```

8.3.6 Using arrays as method arguments and return values

An array can be used as a method argument, as shown in the `main` method where the argument is an array of strings. When the method is called, a reference to the array becomes a local variable within the method. This means that it is sometimes possible to modify an array object from within the method.

For example, if the array is of primitive type then the array elements can be modified. If the array is of object type then the array references can be used to modify the associated objects if they are mutable.

■ EXAMPLE 8.20 **(Array sum method)** In Example 8.10 we wrote statements to sum the elements of an array. It is easy to write a method called `sum` that takes a `double` array as a formal argument, calculates the sum of the elements and returns it. The method prototype is

```
public double sum(double[] a)
```

The square brackets indicate that `a` is an array of double precision numbers. The complete method declaration is

```
public double sum(double[] a)
{
   double s = 0.0;
   for (int k = 0; k < a.length; k++)
   {
      s = s + a[k];
   }
   return s;
}
```

The array reference `a` becomes a local variable inside the method. ■

■ EXAMPLE 8.21 **(Method that prints an array)** The method

```
public void printArray(int[] a)
{
   System.out.print("<" + a[0]);
   for (int k = 1; k < a.length; k++)
   {
      System.out.print("," + a[k]);
   }
   System.out.print(">");
}
```

can be used to print an integer array in the format `<1,2,3>`. ■

■ EXAMPLE 8.22 **(Method to connect an array of points with lines)** The method

```
public void drawLines(Graphics2D g2D, Point2D.Double[] p)
{
   for (int k = 0; k < p.length - 1; k++)
   {
      g2D.draw(new Line2D.Double(p[k], p[k+1]));
   }
}
```

based on Example 8.17 can be used to connect the points with lines. ■

■ EXAMPLE 8.23 **(Modifying elements of array arguments)** To show that array elements can be modified by a method, when the array is an argument, consider the following method

```
public void timesTwo(int[] a)
{
   for (int k = 0; k < a.length; k++)
   {
      a[k] = 2 * a[k];
   }
}
```

which multiplies the elements of an integer array by 2. If you call this method using statements such as

```
int[] myArray = {1,2,3,4,5};
timesTwo(myArray);
for (int k = 0; k < myArray.length; k++)
{
   System.out.println(myArray[k]);
}
```

then the numbers printed are 2, 4, 6, 8, and 10, indicating that the array elements in myArray were changed by the method. This is what is expected since inside the method a and myArray both reference the same array. ■

■ EXAMPLE 8.24 **(Reading an array interactively and returning it)** The method

```
public int[] readArray()
{
   Scanner input = new Scanner(System.in);
   System.out.print("Enter size of array: ");
   int size = input.nextInt();
   input.nextLine();
   int[] a = new int[size];
   for (int k = 0; k < a.length; k++)
   {
      System.out.print("Enter element " + k + ": ");
      a[k] = input.nextInt();
      input.nextLine();
   }
   return a;
}
```

can be used to read an array using console input. The return type indicates that the method returns a reference to an array. To call the method from within the same class use a statement such as

```
int[] testArray = readArray();
```

The method is responsible for reading the array size from the user, constructing the array, asking the user for the array elements, and returning a reference to the array. It is important to realize that a is a local variable, so it disappears when the method exits. However, the array is an object and it doesn't disappear since we are returning a reference to it as the value of the method and assigning it to testArray for example. ■

Testing an array processing method

The following simple class can be used to test an average method for finding the average of the numbers in an array.

Class Average

book-projects/chapter8/array_average

```
package chapter8.array_average; // remove this line if you're not using packages
/**
 * A simple class for testing the array average method
 */
public class Average
{
   /**
    * Return the average of the elements of an array.
    * @param a the array to average
    * @return the average of the elements of array a
    */
   public double average(double[] a)
   {
      double s = 0;
      for (int k = 0; k < a.length; k++)
      {
         s = s + a[k];
      }
      return s / (double) a.length;
   }
}
```

You can easily test this in BlueJ using the following steps.

1. Construct an `Average` object called `avg`.
2. From its method menu choose `average`.
3. Enter an array initializer such as `{1,2,3,4,5}` in the input text box and the average will be shown in the method result window.

To test the method from the command line we can use command-line arguments to enter the array as shown in the following class.

Class `AverageRunner`

book-projects/chapter8/array_average

```
package chapter8.array_average; // remove this line if you're not using packages
/**
 * Command line tester for the Average class.
 * Get array as command-line args.
*/
public class AverageRunner
{
   public static void main(String[] args)
   {
      double[] a = new double[args.length];  // construct array
      for (int k = 0; k < args.length; k++)  // get its elements
      {
         a[k] = Double.parseDouble(args[k]); // convert string to double
      }
      Average avg = new Average();
      System.out.println("Average is " + avg.average(a));
   }
}
```

Some typical output is

```
      java AverageRunner 1 2 3 4 5
      Average is 3.0
```

```
ALGORITHM FindMaximum($\langle a_0, a_1, \ldots, a_{n-1} \rangle$)
$index \leftarrow 0$
FOR $k \leftarrow 1$ TO $n-1$ DO
    IF $a_k > a_{index}$ THEN
        $index \leftarrow k$
    END IF
END FOR
RETURN $index$
```

Figure 8.3: Pseudo-code algorithm for maximum array element

8.4 Some simple array algorithms

Many algorithms can be expressed using arrays. The simplest ones are for finding the maximum and minimum values in an array.

Searching and sorting are two of the most important processing operations performed by computers so it is important to have efficient algorithms. Here we consider only the array version of the simplest searching algorithm, called linear search, for finding a given element in an array.

Then we consider the array version of the simplest sorting algorithm called bubble sort. In a later Chapter we cover searching and sorting in more detail using more efficient algorithms.

8.4.1 Algorithm for the maximum array element

The maximum problem can be stated as follows:

> "Given the array $\langle a_0, \ldots, a_{n-1} \rangle$, determine an index i such that $0 \le i \le n-1$ and $a_i \ge a_k$ for all k such that $0 \le k \le n-1$. Then the maximum value is a_i."

The index i is not unique since the maximum value may occur more than once.

The algorithm begins by assuming the maximum value is at index 0, then a loop is used to process the remaining elements in the array. Each time a larger value is obtained the index is updated. The pseudo-code algorithm is given in Figure 8.3. The final value of *index* is such that a_{index} is the maximum value. This algorithm finds the first occurrence of the maximum. To find the last occurrence change the inequality to $a_k \ge a_{index}$. We could have written the algorithm to directly return the maximum value instead of the index but this would be less general since the position of the maximum would be unknown.

A similar pseudo-code algorithm for finding the minimum can be written by changing the comparison $a_k > a_{index}$ to $a_k < a_{index}$.

Here is a simple tester class for the Java implementation of this algorithm

Class `MaxFinder`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you're not using packages
/**
 * A simple class for testing the findMaximum method
 */
public class MaxFinder
{
   /**
    * Determines the maximum array element and returns its position.
    * @param a the array
    * @return position of the first occurrence of the maximum
    */
   public int findMaximum(double[] a)
   {
      int index = 0;
      for (int k = 1; k <= a.length - 1; k++)
      {
         if (a[k] > a[index])
            index = k;
      }
      return index;
   }
}
```

This class can easily be tested in BlueJ as described above for the `Average` class. The following example shows how to test it in BeanShell.

■ EXAMPLE 8.25 **(Testing `findMaximum` in BeanShell)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter8/array_algorithms");
bsh % MaxFinder f = new MaxFinder();
bsh % double[] a = new double[] {1,2,5,4,3};
bsh % int maxIndex = f.findMaximum(a);
bsh % print(maxIndex);
2
bsh % print(a[maxIndex]);
5
```

The result returned is 2, the index of the maximum value 5. In BeanShell it is necessary to use `new double[]{1,2,5,4,3}` to specify the array argument. ■

To test the class from the command line we can use the simple runner class

Class `MaxFinderRunner`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you're not using packages
/**
 * Command line tester for the MaxFinder class.
 * Get array as command line args.
 */
```

```
public class MaxFinderRunner
{
   public static void main(String[] args)
   {
      double[] a = new double[args.length];  // construct array
      for (int k = 0; k < args.length; k++)  // get its elements
      {
         a[k] = Double.parseDouble(args[k]); // convert string to double
      }
      MaxFinder finder = new MaxFinder();
      int indexMax = finder.findMaximum(a);
      System.out.println("Index of maximum is " + indexMax);
      System.out.println("Maximum value is " + a[indexMax]);
   }
}
```

that uses command-line arguments.

We have used an array of `double` numbers but the algorithm is easily modified to use an array of any type for which the elements can be ordered and compared.

The following example gives the algorithm for an array of `BankAccount` objects where the ordering is defined by the account balance.

■ EXAMPLE 8.26 **(Maximum bank account balance method)** The method

```
public int findMaximum(BankAccount[] a)
{
   int index = 0;
   for (int k = 1; k <= a.length - 1; k++)
   {
      if (a[k].getBalance() > a[index].getBalance())
         index = k;
   }
   return index;
}
```

returns the array index of the reference to the bank account that has the maximum balance. ■

Similarly, the following example gives the algorithm for an array of strings.

■ EXAMPLE 8.27 **(String array example)** The method

```
public int findMaximum(String[] a)
{
   int index = 0;
   for (int k = 1; k <= a.length - 1; k++)
   {
      if (a[k].compareTo(a[index]) > 0)
         index = k;
   }
   return index;
}
```

```
**ALGORITHM** LinearSearch ( $\langle a_0, a_1, \ldots, a_{n-1} \rangle, x$ )
$index \leftarrow 0$
**WHILE** $(index \leq n-1) \wedge (a_{index} \neq x)$ **DO**
    $index \leftarrow index + 1$
**END WHILE**
**IF** $index > n-1$ **THEN**
    **RETURN** $-1$
**ELSE**
    **RETURN** $index$
**END IF**
```

Figure 8.4: Pseudo-code linear search algorithm

returns the array index of the reference to the string that lexicographically follows all strings in the array using the `compareTo` method. ■

8.4.2 Linear search slgorithm

In a linear search of an array we are looking for a given value x among the array elements. If we find x we can return its index. Otherwise we can return the invalid index -1. The linear search problem can be stated as follows:

> "Given the array $\langle a_0, \ldots, a_{n-1} \rangle$ and a value x to find, determine an index i such that $a_i = x$ and $0 \leq i \leq n-1$. If such an index cannot be found let the index be -1."

We cannot use a for-loop here since we do not know how many times the body of the loop will be executed. The loop continues as long as there are elements in the array left to examine and as long as we have not found the element we are looking for. Therefore, we use a while-loop.

The pseudo-code algorithm is given in Figure 8.4. There are two ways the while-loop can terminate. If $index \leq n-1$ is false we have "gone off the end" of the array and the entire boolean expression is false so the loop will exit. Because of short-circuit evaluation, the expression $a_{index} \neq x$ will not be evaluated in this case. Otherwise the array index could be out of range. If the element x is found then the expression $a_{index} \neq x$ will be false and the loop will exit. When the loop exits we can test *index* to see which exit was taken. If $index > n-1$ then we did not find x so -1 is returned. Otherwise, x was found and *index* is returned.

It is easy to translate this pseudo-code algorithm into the following tester class for the corresponding Java method.

Class `LinearSearcher`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you're not using packages
/**
```

```
 * A simple class for testing the linearSearch method
 */
public class LinearSearcher
{
   /**
    * Find a given element in an array
    * @param a the array
    * @param x the element to search for
    * @return position of the first occurrence of x or -1
    * if x is not found.
    */
   public int search(double[] a, double x)
   {
      int index = 0;
      int n = a.length; // number of array elements
      while (index < n && a[index] != x)
      {
         index = index + 1;
      }
      if (index >= n)
         return -1;
      else
         return index;
   }
}
```

This class can be tested using BlueJ and the following example shows how to test the class using BeanShell

■ EXAMPLE 8.28 **(Testing linear search using BeanShell)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter8/array_algorithms");
bsh % LinearSearcher searcher = new LinearSearcher();
bsh % int index = searcher.search( new double[]{1,2,3,4,5}, 1);
bsh % print(index);
0
bsh % int index = searcher.search( new double[]{1,2,3,4,5}, 2);
bsh % print(index);
1
bsh % int index = searcher.search( new double[]{1,2,3,4,5}, 7);
bsh % print(index);
-1
```

to test the linear search algorithm. The final result shows that 7 was not found in the array. ■

A runner class for command-line testing is given by

Class `LinearSearcherRunner`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you’re not using packages
/**
 * Command line tester for the LinearSearch class.
 * Get array as command line args. Last command line
 * argument is the element to search for
*/
public class LinearSearcherRunner
{
   public static void main(String[] args)
   {
      double[] a = new double[args.length - 1];  // construct array
      for (int k = 0; k < args.length - 1; k++)  // get its elements
      {
         a[k] = Double.parseDouble(args[k]); // convert string to double
      }
      double x = Double.parseDouble(args[args.length - 1]); // last argument is x

      LinearSearcher searcher = new LinearSearcher();
      int index = searcher.search(a, x);
      System.out.println("Index of element is " + index);
   }
}
```

8.4.3 Bubble sort algorithm

There are many sorting algorithms. The simplest is called bubble sort. It is not very efficient for large arrays but it is the easiest to understand. More efficient algorithms will be considered in a later Chapter.

For arrays the basic sorting problem is to rearrange the elements in increasing order, or in decreasing order. For example, the array $\langle 5,3,8,5,4,2,2\rangle$ is not sorted. In increasing order the sorted array is $\langle 2,2,3,4,5,5,8\rangle$. Similarly, the string array $\langle$”one”, ”two”, ”three”$\rangle$ is not sorted. In increasing lexicographic order the sorted array is $\langle$”one”, ”three”, ”two”$\rangle$.

Bubble sort is the easiest sorting algorithm to understand because of its intuitive nature. If the elements of $\langle a_0,\ldots,a_{n-1}\rangle$ can be ordered the algorithm for increasing order is

- **Pass** 1: Process the array elements a_0 to a_{n-1} exchanging or swapping elements that are out of order: if $a_0 > a_1$, swap them, if $a_1 > a_2$ swap them, ..., if $a_{n-2} > a_{n-1}$ swap them. After this first pass through the array the largest element will be in the last position, its correct position. In other words the largest element “bubbles to the top” so we don’t need to consider it again.

- **Pass** 2: For the second pass process the elements a_0 to a_{n-2}, swapping elements that are out of order. At the end of this pass the elements a_{n-2} and a_{n-1} are in their correct positions.

- **Pass** $n-1$**:** For the final pass a_2 to a_{n-1} are in their correct position so we need only consider a_0 and a_1: If $a_0 > a_1$ then swap these elements

If we denote the pass number by p then after pass 1 one element is in its correct position, after pass 2 two elements are in their correct position and after $n-1$ passes $n-1$ elements are in their correct position. We can stop here since if $n-1$ elements are in their correct position then the only remaining element, a_0 must be in its correct position as the smallest array element. This means that the outer loop over the pass number ranges from $p = 1$ to $p = n-1$.

At the start of pass number p we have the $p-1$ elements $a_{n-1-(p-1)}$ to a_{n-1} in their correct positions so we need to compare the elements a_0 to a_{n-1-p}. Thus, the inner loop goes from $j = 0$ to $j = n-1-p$. This gives the top level pseudo-code algorithm

```
**FOR** $p \leftarrow 1$ **TO** $n-1$ **DO**
    Compare pairs at positions $(0,1),(1,2),\ldots,(n-1-p,n-p)$
    swapping elements that are out of order.
**END FOR**
```

As an example, consider the array $\langle a_0,\ldots,a_7\rangle$ given by $\langle 44,55,12,42,94,18,6,67\rangle$. The steps are shown in Table 8.1, where boldface elements are in their correct position. A pseudo-code algorithm

Pass	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Start of pass 1	44	55	12	42	94	18	6	67
Start of pass 2	44	12	42	55	18	6	67	**94**
Start of pass 3	12	42	44	18	6	55	**67**	**94**
Start of pass 4	42	12	18	6	44	**55**	**67**	**94**
Start of pass 5	12	18	6	42	**44**	**55**	**67**	**94**
Start of pass 6	12	6	18	**42**	**44**	**55**	**67**	**94**
Start of pass 7	6	12	**18**	**42**	**44**	**55**	**67**	**94**
End of pass 7	6	**12**	**18**	**42**	**44**	**55**	**67**	**94**

Table 8.1: Bubble sort example

for selection sort is given in Figure 8.5.

```
**ALGORITHM** bubbleSort($\langle a_0,a_1,\ldots,a_{n-1}\rangle$)
**FOR** $p \leftarrow 1$ **TO** $n-1$ **DO**
    **FOR** $j \leftarrow 0$ **TO** $n-1-p$ **DO**
        **IF** $a_j > a_{j+1}$ **THEN**
            swap($a_j,a_{j+1}$)
        **END IF**
    **END FOR**
**END FOR**
```

Figure 8.5: Pseudo-code bubble sort algorithm

Sorting an array of numbers

It is easy to translate this pseudo-code algorithm into the following tester class for the corresponding Java method.

Class `BubbleSorter`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you're not using packages
/**
 * A simple class for testing the bubbleSort method.
 * We use the double[] return type instead of void
 * for the bubbleSort method so the method can be
 * directly tested with BlueJ.
 */
public class BubbleSorter
{
   /**
    * Sort an array in increasing order
    * @param a the array
    */
   public double[] bubbleSort(double[] a)
   {
      int n = a.length;
      for (int p = 1; p <= n - 1; p++) // loop over passes
      {
         for (int j = 0; j <= n - 1 - p; j++)
         {
            if (a[j] > a[j + 1])
            {
               double temp = a[j];
               a[j] = a[j + 1];
               a[j + 1] = temp;
            }
         }
      }
      return a;
   }
}
```

Three assignment statements are needed to exchange (swap) two values, since it is necessary to use a temporary variable to save the first element of the pair being swapped.

Normally the return type of the `bubbleSort` method would be `void` but we have made it `double[]` here so that the class can be tested using BlueJ as follows

1. Construct a `BubbleSorter` object.
2. From its object menu select the `bubbleSort` method
3. In the resulting "method call" window enter an array such as `{5,4,1,6,2}` in the text box.
4. You will get a "Method Result" window showing the array as an `<object reference>` so click on it and then click the "Inspect" button to see the sorted array or click the "Get" button to put a reference to the sorted array on the workbench.

The following example shows how to test the class using BeanShell

■ EXAMPLE 8.29 **(Testing bubble sort using BeanShell)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter8/array_algorithms");
bsh % BubbleSorter sorter = new BubbleSorter();
bsh % double[] a = {44,55,12,42,94,18,6,67};
bsh % sorter.bubbleSort(a);
bsh % print(a);
double[]: {6.0,12.0,18.0,42.0,44.0,55.0,67.0,94.0}
```

to test the bubble sort algorithm using the example given in Table 8.1 ■

A runner class for command-line testing is given by

Class `BubbleSorterRunner`

book-projects/chapter8/array_algorithms

```
package chapter8.array_algorithms; // remove this line if you're not using packages
/**
 * Command line tester for the BubbleSorter class.
 * Get array as command line args.
 */
public class BubbleSorterRunner
{
   public static void main(String[] args)
   {
      double[] a = new double[args.length];  // construct array
      for (int k = 0; k < args.length; k++)  // get its elements
      {
         a[k] = Double.parseDouble(args[k]); // convert string to double
      }
      BubbleSorter sorter = new BubbleSorter();
      System.out.println("Array to sort is " + arrayToString(a));
      sorter.bubbleSort(a);
      System.out.println("Sorted array is  " + arrayToString(a));
   }

   public static String arrayToString(double[] a)
   {
      String s = "<" + a[0];
      for (int k = 1; k < a.length; k++)
         s = s + "," + a[k];
      s = s + ">";
      return s;
   }
}
```

Here we have included an `arrayToString` method to make a string representation of an array which can be displayed. Some typical output is

```
java BubbleSorterRunner 44 55 12 42 94 18 6 67
Array to sort is <44.0,55.0,12.0,42.0,94.0,18.0,6.0,67.0>
Sorted array is  <6.0,12.0,18.0,42.0,44.0,55.0,67.0,94.0>
```

Sorting an array of strings

We have written the bubble sort method to sort an array of type `double[]` but it can easily be modified to sort arrays of other types. The following example gives a method for sorting a string array.

■ EXAMPLE 8.30 **(Sorting an array of strings)** The method

```
public void bubbleSort(String[] a)
{
   int n = a.length;
   for (int p = 1; p <= n - 1; p++) // loop over passes
   {
      for (int j = 0; j <= n - 1 - p; j++)
      {
         if (a[j].compareTo(a[j+1]) > 0)
         {
            String temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
         }
      }
   }
}
```

sorts an array of strings in increasing lexicographical order.

Recall that a string array is not an array of string objects, it is an array of references to string objects. Therefore the method does not swap string objects in memory. This would be very inefficient. Instead string references are swapped. The final result is a sorted array of string references such that the string referenced by `a[0]` precedes the string referenced by `a[1]`, and so on. ■

8.5 Efficient evaluation of a polynomial

Polynomials are simple functions that occur in many applications. They can be represented by arrays and we want to develop an efficient algorithm to evaluate them. A polynomial of degree n is a function p defined for each value of x by

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

in terms of the array $\langle a_0, a_1, \ldots, a_n \rangle$ of $n+1$ coefficients with $a_n \neq 0$. An “obvious” way to calculate $p(x)$, given x and the coefficient array, is

```
$sum \leftarrow a_0$
FOR $k \leftarrow 1$ TO $n$ DO
    $sum \leftarrow sum + a_k x^k$
END FOR
```

```
**ALGORITHM** EvaluatePolynomial($\langle a_0, a_1, \ldots, a_n\rangle$, $x$ )
$p \leftarrow a_n$
**FOR** $k \leftarrow n-1$ **TO** 0 **BY** $-1$ **DO**
    $p \leftarrow a_k + xp$
**END FOR**
**RETURN** $p$
```

Figure 8.6: Pseudo-code polynomial evaluation algorithm

However, this is not a very efficient algorithm: n multiplications to compute $a_n x^n$, $n-1$ multiplications to compute $a_{n-1}x^{n-1}$, and so on. The total number of multiplications is

$$n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}.$$

8.5.1 Horner's algorithm

We can compute $p(x)$ with only n multiplications using Horner's algorithm. To derive it, write the polynomial in the nested form

$$\begin{aligned} p(x) &= a_0 + x(a_1 + a_2 x + \cdots + a_n x^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + \cdots + a_n x^{n-2})) \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1} + x a_n)) \cdots)) \end{aligned}$$

and evaluate it from the "inside out" using the following scheme:

$$\begin{aligned} p_n &= a_n \\ p_{n-1} &= a_{n-1} + x a_n = a_{n-1} + x p_n \\ p_{n-2} &= a_{n-2} + x(a_{n-1} + x a_n) = a_{n-2} + x p_{n-1} \\ \cdots & \\ p_k &= a_k + x p_{k+1} \\ \cdots & \\ p_1 &= a_1 + x p_2 \\ p_0 &= a_0 + x p_1 \end{aligned}$$

Then p_0 is the value of $p(x)$. In the loop the subscript k moves downward from $n-1$ to 0 with each iteration. However, we do not need to use subscripts on p since each value can be obtained from the previous one using $p \leftarrow a_k + xp$. The elegant pseudo-code algorithm shown in Figure 8.6 evaluates the polynomial using only n multiplications (the for-loop executes n times and there is one multiplication each time).

8.5.2 A class for polynomials

We can think of a polynomial as an object from a class called `Polynomial`. Since each polynomial is defined by its coefficient array then a reference to this array can be the private data field so we need a constructor with an array argument. Also we include an `eval` method based on the pseudo-code algorithm for evaluating a polynomial:

Class `Polynomial`

book-projects/chapter8/polynomial

```
package chapter8.polynomial; // remove this line if you're not using packages
/**
 * A class that represents a polynomial using an array and
 * shows how to use Horner's rule to efficiently evaluate
 * a polynomial p. A reference to the user's array is kept
 * as a data field and an eval method is
 * provided for evaluating p at a given value of x.
 */
public class Polynomial
{
   private double[] a; // the coefficient array

   /**
    * Construct a polynomial p with a given coefficient array
    * @param coefficients the array of coefficients such that
    * coefficients[k] is the coefficient of x^k
    */
   public Polynomial(double[] coefficients)
   {
      a = coefficients;
   }

   /**
    * Evaluate the polynomial p.
    * @param x the value to evaluate polynomial at.
    * @return p(x) the value of p at x
    */
   public double eval(double x)
   {
      int n = a.length - 1;
      double p = a[n];
      for (int k = n-1; k >= 0; k--)
      {
         p = a[k] + x*p;
      }
      return p;
   }

   /**
    * Return a string representation of a polynomial in format
    * a[0] + a[1]x + a[2]x^2 + ...
    */
   public String toString()
   {
      String p = (a[0] == 0) ? "" : "" + a[0];

      for (int k = 1; k < a.length; k++)
      {
         String term = (k == 1) ? "x" : "x^" + k;

         if (p.equals("") && a[k] != 0)
         {
            p = a[k] + term;
         }
         else
         {
            if (a[k] > 0)
            {
               p = p + " + " + a[k] + term;
            }
            else if ( a[k] < 0 )
            {
               p = p + " - " + Math.abs(a[k]) + term;
            }
         }
      }
      if (p.equals("")) p = "0";
      return p;
   }
}
```

The `toString` method returns a string representation of the polynomial using `x^n` to represent x^n. It is a little complicated because there are several cases. For example, if there is a first power we want to display just `x`, not `x^1`.

To construct the polynomial p defined by $p(x) = 1 + 2x + 3x^2$ we can use statements such as

```
double[] coeff = new double[] {1,2,3};
Polynomial p = new Polynomial(coeff);
```

or even the single statement

```
Polynomial p = new Polynomial(new double[] {1,2,3});
```

Now to evaluate $p(3.5)$ we can use a statement such as

```
double val = p.eval(3.5);
```

8.5.3 Testing the `Polynomial` class

To test the `Polynomial` class in BlueJ perform the following steps.

1. Construct a `Polynomial` object and enter an array such as `{1,0,3,0,5}` in the input box.

2. From its object menu select the `eval` method and enter `1.5`.
3. In the resulting "Method Result" window the value `33.0625` is shown.
4. From the object menu select the `toString` method to see `"1.0 + 3.0xˆ2 + 5.0xˆ4"` in the "Method Result" window.

The following example shows how to test the class using BeanShell.

■ EXAMPLE 8.31 **(Evaluating polynomials using BeanShell)** Try the statements

```
bsh % addClassPath("c:/book-projects/chapter8/polynomial");
bsh % Polynomial p = new Polynomial(new double[]{1,0,3,0,5});
bsh % print(p.eval(1.5));
33.0625
bsh % print(p);
1.0 + 3.0xˆ2 + 5.0xˆ4
bsh %
```

to test the `Polynomial` class for the example $p(x) = 1 + 3x^2 + 5x^4$. ■

To test the `Polynomial` class here is a runner class that gets the coefficients of a polynomial followed by the value of x as command-line arguments, constructs the polynomial, and displays it and its value at x.

Class `PolynomialRunner`

book-projects/chapter8/polynomial

```
package chapter8.polynomial; // remove this line if you're not using packages
/**
 * Class to test Polynomial using command-line arguments.
 * If there are n arguments the first n-1 define the coefficient
 * array and the last one gives the value of x
 */
public class PolynomialRunner
{
   public static void main(String[] args)
   {
      int n = args.length;

      if (args.length >= 2)
      {
         // First n-1 args are the coefficients so construct
         // a polynomial using them

         double[] coefficients = new double[n-1];
         for (int k = 0; k < n-1; k++)
         {
            coefficients[k] = Double.parseDouble(args[k]);
         }
         Polynomial p = new Polynomial(coefficients);

         System.out.println("p(x) = " + p);

         // Last argument is the value of x

         double x = Double.parseDouble(args[n-1]);

         // Evaluate the polynomial at x

         System.out.println("p(" + x + ") = " + p.eval(x));
      }
      else
      {
         System.out.println("args: a0 a1 ... an x");
      }
   }
}
```

Here is some output for evaluating the polynomial $p(x) = 1 + 3x^2 + 5x^4$ at $x = 1.5$ and $x = 3.5$.

```
java PolynomialRunner 1 0 3 0 5 1.5
p(x) = 1.0 + 3.0x^2 + 5.0x^4
p(1.5) = 33.0625

java PolynomialRunner 1 0 3 0 5 3.5
p(x) = 1.0 + 3.0x^2 + 5.0x^4
p(3.5) = 788.0625
```

8.6 Line graph example using arrays

Many graphics programs require arrays to store points and lines. In this section we develop a `LineGraph` class for representing line graphs. The class has an array of points as an instance data field and draws line segments from one point to the next to obtain a line graph.

Another example is a bar graph class that stores an array of height values for the bars in a vertical bar graph and draws the bars (see Exercise 8.6). In each case the array data can be used to calculate the bounding box of the graph so that an appropriate coordinate system can be chosen.

8.6.1 Line graph class

We want to draw a line graph given an array $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ of n vertices $v_k = (x_k, y_k)$ specified in order of increasing x-coordinate. Each pair of vertices is to be joined by a line segment and a small circle should appear at each vertex. An example for seven vertices is shown in Figure 8.7.

We won't assume any particular coordinate system. For example, a line graph can represent a stock market's closing average for several consecutive days or it might represent the daily high temperatures for the month of June.

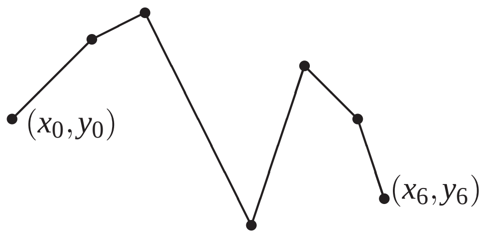


Figure 8.7: Line graph for vertex array $\langle(x_0,y_0),(x_1,y_1),\ldots,(x_6,y_6)\rangle$

Instead, to make the graph fit in the drawing window we will use the array data to compute the bounding box of the graph in the world coordinate system. Then we can use the `worldTransform` method from the `BarGraph3` class (Chapter 5, page 239) to transform world coordinates to default user coordinates. The `LineGraph` class will have the following specification.

```
public class LineGraph extends JPanel
{
   private Point2D.Double[] v; // vertices of line graph
   private double xMin, xMax, yMin, yMax; // bounding box of graph

   /* Constructor for a given vertex array p */
   public LineGraph(Point2D.Double[] p) {...}

   public void paintComponent(Graphics g) {...}

   /* Compute xMin, xMax, yMin, yMax */
   private void computeBoundingBox() {...}

   /* Perform the affine transformation */
   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h) {...}
}
```

To make the class more reusable we do not input the vertices here. Instead we leave this to the user of the class. A `LineGraph` object receives a reference to the the vertex array as the constructor argument. Each `LineGraph` object represents a line graph such as the one shown in Figure 8.8.

Choosing a coordinate system

The private method `computeBoundingBox` needs to determine the smallest and largest of the *x*-coordinates in the vertex array and similarly for the *y*-coordinates. This can be done in one loop as follows

```
private void computeBoundingBox()
{
   xMin = v[0].getX();
   xMax = v[0].getX();
```

```
      yMin = v[0].getY();
      yMax = v[0].getY();
      for (int k = 1; k < v.length; k++)
      {
         double x = v[k].getX();
         double y = v[k].getY();
         if (x < xMin) xMin = x;
         if (x > xMax) xMax = x;
         if (y < yMin) yMin = y;
         if (y > yMax) yMax = y;
      }
   }
```

We want to define a coordinate system slightly larger than this bounding box to allow for a 5% border around the graph as shown in Figure 8.8. This is done in the `paintComponent` method as follows.

```
   int w = getWidth();  // JPanel width in pixels
   int h = getHeight(); // JPanel height in pixels
   double bx = (xMax - xMin) * 0.05;
   double by = (yMax - yMin) * 0.05;
   AffineTransform world =
      new worldTransform(xMin-bx, xMax+bx, yMin-by, yMax+by, w, h);
   g2D.transform(world);
   double pixelWidth  = Math.abs(1 / world.getScaleX());
   double pixelHeight = Math.abs(1 / world.getScaleY());
   float thickness = (float) Math.min(pixelWidth, pixelHeight);
   g2D.setStroke(new BasicStroke(thickness));
```

where we have also determined the dimensions of one pixel in the world coordinate system and set the line width to one pixel.

Drawing the axes

If $yMin \leq 0 \leq yMax$ the x-axis will be visible and similarly if $xMin \leq 0 \leq xMax$ the y-axis will be visible. Therefore to draw the axes we use

```
   g2D.setPaint(Color.blue);
   if (yMin <= 0.0 && 0.0 <= yMax)
   {
      g2D.draw(new Line2D.Double(xMin,0,xMax,0));
   }
   if (xMin <= 0.0 && 0.0 <= xMax)
   {
      g2D.draw(new Line2D.Double(0,yMin,0,yMax));
   }
```

Drawing the line segments

If there are n vertices $v_0, \ldots, v_{n-1}$ then the first line segment joins v_0 to v_1 and the last joins v_{n-2} to v_{n-1}. Therefore the following loop draws the line segments.

```
g2D.setPaint(Color.black);
for (int k = 0; k < v.length - 1; k++)
{
   double x1 = v[k].getX();
   double y1 = v[k].getY();
   double x2 = v[k+1].getX();
   double y2 = v[k+1].getY();
   g2D.draw(new Line2D.Double(x1,y1,x2,y2));
}
```

Drawing the circles at each vertex

If (x, y) is a vertex we want to draw a filled circle with radius 3 pixels centered at this vertex. The following for-loop draws the circles.

```
double xr = 3 * pixelWidth;
double yr = 3 * pixelHeight;
g2D.setPaint(Color.red);
for (int k = 0; k < v.length; k++)
{
   double x = v[k].getX();
   double y = v[k].getY();
   Ellipse2D.Double ellipse = new Ellipse2D.Double(x-xr, y-yr, 2*xr, 2*yr);
   g2D.fill(ellipse);
}
```

The bounding box of the ellipse has bottom left corner at `(x-xr,y-yr)` and its width and height are `2*xr` and `2*yr`, respectively. Putting everything together we obtain the following class.

Class `LineGraph`

book-projects/chapter8/line_graph

```
package chapter8.line_graph; // remove this line if you're not using packages
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Draw a line graph given its vertices (x,y).
 * The axes are drawn if they are visible. A small circle is also shown
 * at each vertex. Affine transformations are used to map the world
 * coordinate system of the line graph to the device coordinate system.
 */
```

```
public class LineGraph extends JPanel
{
   private Point2D.Double[] v; // vertices of line graph
   private double xMin, xMax, yMin, yMax; // bounding box of graph

   /* Construct a line graph for a specified array of points
   */
   public LineGraph(Point2D.Double[] p)
   {
      v = p;
      computeBoundingBox();
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      int w = getWidth();  // JPanel width in pixels
      int h = getHeight(); // JPanel height in pixels

      // Make a world coordinate system with a 5 percent border

      double bx = (xMax - xMin) * 0.05;
      double by = (yMax - yMin) * 0.05;

      AffineTransform world = worldTransform(xMin-bx, xMax+bx, yMin-by, yMax+by, w, h);
      g2D.transform(world);

      // Width of a pixel in world space is  (xMax - xMin + 2*bx) / (w-1)
      // Height of a pixel in world space is (yMax - yMin + 2*by) / (h-1)
      // But we can get these values from the affine transformation:

      double pixelWidth  = Math.abs(1 / world.getScaleX()); // pixel width in world
      double pixelHeight = Math.abs(1 / world.getScaleY()); // pixel height in world

      // Now we can calculate a line thickness relative that is two pixels wide

      float thickness = (float) Math.min(pixelWidth, pixelHeight);
      g2D.setStroke(new BasicStroke(thickness));

      // Draw the x-axis in blue if it is visible

      g2D.setPaint(Color.blue);
      if (yMin <= 0.0 && 0.0 <= yMax)
      {
         g2D.draw(new Line2D.Double(xMin,0,xMax,0));
      }

      // Draw the y-axis in blue if it is visible
      if (xMin <= 0.0 && 0.0 <= xMax)
      {
         g2D.draw(new Line2D.Double(0,yMin,0,yMax));
      }

      // draw the line segments connecting the vertices in black

      g2D.setPaint(Color.black);
      for (int k = 0; k < v.length - 1; k++)
      {
         double x1 = v[k].getX();
         double y1 = v[k].getY();
         double x2 = v[k+1].getX();
         double y2 = v[k+1].getY();
         g2D.draw(new Line2D.Double(x1,y1,x2,y2));
      }

      // We need to draw a small red circle
      // about each point that has a radius of 3 pixels.

      double xr = 3 * pixelWidth; // ellipse radius in x direction
      double yr = 3 * pixelHeight; // ellipse radius in y direction

      g2D.setPaint(Color.red);
      for (int k = 0; k < v.length; k++)
      {
         double x = v[k].getX();
         double y = v[k].getY();
         Ellipse2D.Double ellipse = new Ellipse2D.Double(x-xr, y-yr, 2*xr, 2*yr);
         g2D.fill(ellipse);
      }
   }

   /* Compute the bounding box of the line graph.
      The x range will be xMin <= x <= xMax
      The y range will be yMin <= y <= yMax
   */
   private void computeBoundingBox()
   {
      xMin = v[0].getX();
      xMax = v[0].getX();
      yMin = v[0].getY();
      yMax = v[0].getY();
      for (int k = 1; k < v.length; k++)
      {
         double x = v[k].getX();
         double y = v[k].getY();
         if (x < xMin) xMin = x;
         if (x > xMax) xMax = x;
         if (y < yMin) yMin = y;
         if (y > yMax) yMax = y;
      }
   }

   private AffineTransform worldTransform(double xMin, double xMax,
      double yMin, double yMax, int w, int h)
   {
      double sx = (w-1) / (xMax - xMin); // scale factor in x direction
      double sy = (h-1) / (yMax - yMin); // scale factor in y direction
      AffineTransform at = new AffineTransform();
      at.scale(sx, -sy);     // -sy reverses y axis
      at.translate(-xMin, -yMax);  // upper left corner (xMin,yMax) to (0,0)
      return at;
   }
}
```

Here is a simple class that uses the `GraphicsFrame` class to draw a graph of 8 vertices.

Class `SimpleTester`

book-projects/chapter8/line_graph

```
package chapter8.line_graph; // remove this line if you’re not using packages
import custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.geom.*;

/**
 * Runner class for testing the LineGraph class
 * by drawing a sample graph if 8 vertices.
 */
public class SimpleTester
{
   public void runTest()
   {
      Point2D.Double[] v = new Point2D.Double[8];
      v[0] = new Point2D.Double(-4, -1);
      v[1] = new Point2D.Double(-3, 1);
      v[2] = new Point2D.Double(-2, 0.5);
      v[3] = new Point2D.Double(-1, 2);
      v[4] = new Point2D.Double(1, 0.5);
      v[5] = new Point2D.Double(3, 3);
      v[6] = new Point2D.Double(4, 0.75);
      v[7] = new Point2D.Double(6, 0.6);

      new GraphicsFrame("A Simple Line Graph", new LineGraph(v), 401, 301);
   }

   public static void main(String[] args)
   {
      new SimpleTester().runTest();
   }
}
```

The output window is shown in Figure 8.8. The line graph is black, the small circles at each vertex

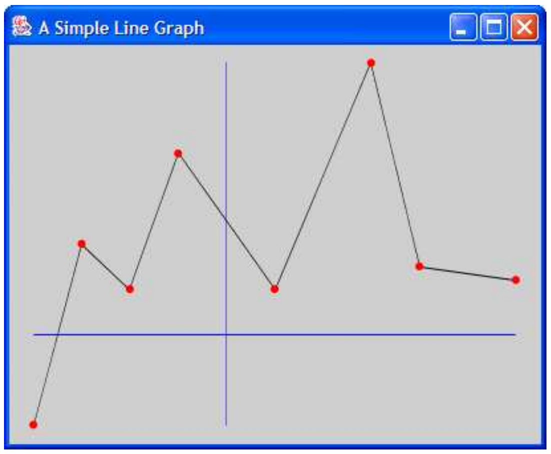


Figure 8.8: Line graph from `SimpleTester`

are red and the coordinate axes $x = 0$ and $y = 0$ are both visible.

8.6.2 Drawing a random line graph

As another example we use the `LineGraph` class to generate a random array of points. The only problem here is that the x-coordinates must increase, $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$, and this won't be the case. However, we can use a version of the `bubbleSort` method (page 404) to sort the vertex array of randomly generated `Point2D.Double` objects in order of increasing x-coordinate before drawing the graph.

We want the number of points in the vertex array to be a random integer in the range 5 to 100, which can be done with the statements

```
int numVertices = (int) (96.0 * Math.random()) + 5;
Point2D.Double[] v = new Point2D.Double[numVertices];
```

recalling that the `random` method generates random numbers r such that $0 \leq r < 1$. Also let us randomly choose each x or y coordinate to be in the range -10 to 10. This can be done with the for-loop

```
for (int k = 0; k < v.length; k++)
{
   double x = 20.0 * Math.random() - 10.0;
   double y = 20.0 * Math.random() - 10.0;
   v[k] = new Point2D.Double(x, y);
}
```

Now the vertex array can be used to construct a `LineGraph` object that is an argument to a `GraphicsFrame` object. Thus, we have the following class to generate the graph.

Class **RandomTester**

book-projects/chapter8/line_graph

```
package chapter8.line_graph; // remove this line if you’re not using packages
package custom_classes.GraphicsFrame; // remove this line if you’re not using packages
import java.awt.geom.*;

/**
 * Test the LineGraph class by generating a random array of points sorted
 * in order of increasing x coordinate.
 */
public class RandomTester
{
   public void runTest()
   {
     // random number of vertices in range 5 to 100

      int numVertices = (int) (96.0 * Math.random()) + 5;
      Point2D.Double[] v = new Point2D.Double[numVertices];

      // Generate random points with coordinates in range -10 to 10
      // and sort them on increasing x coordinate.

      for (int k = 0; k < v.length; k++)
      {
         double x = 20.0 * Math.random() - 10.0;
         double y = 20.0 * Math.random() - 10.0;
         v[k] = new Point2D.Double(x, y);
      }

      bubbleSort(v);
      new GraphicsFrame("A Random Line Graph", new LineGraph(v), 401, 301);
   }

   /* Sort an array of points in increasing order of x-coordinates.
   */
   private void bubbleSort(Point2D.Double[] a)
   {
      int n = a.length;
      for (int p = 1; p <= n - 1; p++)
      {
         for (int j = 0; j <= n - 1 - p; j++)
         {
            if (a[j].getX() > a[j + 1].getX())
            {
               Point2D.Double temp = a[j];
               a[j] = a[j + 1];
               a[j + 1] = temp;
            }
         }
      }
   }
```

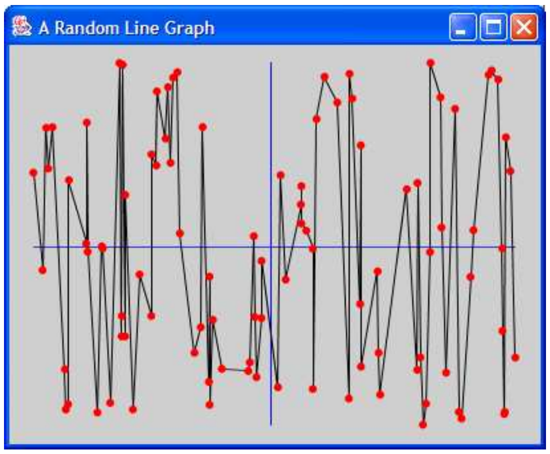


Figure 8.9: A `RandomTester` output window

```
   public static void main(String[] args)
   {
      new RandomTester().runTest();
   }
}
```

A typical output window is shown in Figure 8.9.

8.6.3 Converting arrays to `GeneralPath` objects

In Chapter 5 we used the `GeneralPath` class to construct custom shapes. We can use arrays of points to construct `GeneralPath` objects and then draw or fill them. This technique could have been used in the `LineGraph` class: rather than draw line segments one at a time we could have used the following statements

```
      GeneralPath p = new GeneralPath();
      p.moveTo((float) v[0].getX(), (float) v[0].getY());
      for (int k = 1; k < v.length; k++)
      {
         p.lineTo((float) v[k].getX(), (float) v[k].getY());
      }
      g2D.draw(p):
```

to define the path `p` as a `Shape` object and then draw it.

8.7 For-each loop

In some for loops that sequentially process the elements of an array the only purpose of the loop index is to suceessively reference the next element in the array. For example, the following for loop prints the elements of an array `a`, one per line:

```
for (int k = 0; k < a.length; k++)
{
   System.out.println(a[k]);
}
```

Here the loop index `k` is used only to refer to `a[k]`. What we really need in this case is a loop which says “iterate over the elements of the array from beginning to end and display each array element”.

In Java 5 a special kind of for loop called the “for each” loop was introduced to deal with this situation. It has the syntax

`for` (*Type name* : *collection*)
{
   Statements
}

Here *collection* is the name of a collection of objects of the given *Type* and the variable *name* will successively take on the value of the next element in the collection (the only collection we know at this stage is the array). This kind of for loop is sometimes called an **iterator**. Read the colon as “in” so the loop says “for each name in the collection ...”

For example, the for loop that prints elements can be expressed for an array `a` of type `int[]` as

```
for (int elem : a)
{
   System.out.println(elem);
}
```

This is much nicer. Here the first value of `elem` is `a[0]`, the next value is `a[1]` and so on until the last value is `a[a.length-1]`.

■ EXAMPLE 8.32 **(Summing an integer array)** The two methods

```
public int arraySum(int[] a)               public int arraySum(int[] a)
{                                          {
   int sum = 0;                               int sum = 0;
   for (int k = 0; k < a.length; k++)         for (int elem : a)
   {                                          {
      sum = sum + a[k];                          sum = sum + elem;
   }                                          }
   return sum;                                return sum;
}                                          }
```

show how to sum the elements of an integer array using the ordinary for loop and the new one. ■

■ EXAMPLE 8.33 **(Average balance in a `BankAccount` array)** The method

```
public double averageBalance(BankAccount[] b)
{
   double sum = 0.0;
   for (BankAccount account : b)
   {
      sum = sum + account.getBalance();
   }
   return sum / b.length;
}
```

returns the average bank balance in the given array of bank accounts ■

8.8 Methods with a variable number of arguments

In Java 5 it is possible to have methods with a variable number of arguments. We have already seen two such methods: `String.format` and `printf` (see Chapter 4.2.5, page 102). The syntax for the method prototype is

ReturnType methodName(*initialArgumentList*, *Type* ... *name*)

For example the `String.format` method has prototype

```
public static format(String format, Object ... args)
```

In this case there is one initial argument to specify the format string. It is followed by a variable number of arguments of type `Object`. Here `args` refers to the variable part of the argument list. Within the method it can be accessed as an array using the standard for loop or the for each loop.

■ EXAMPLE 8.34 **(Min method with variable number of arguments)** As an example here is a method that takes a variable number of integer arguments and returns the minimum value among the arguments.

```
public int min(int ... args)
{
   int minValue = Integer.MAX_VALUE;
   for (int k = 0; k < args.length; k++)
   {
      if (args[k] < minValue) minValue = args[k];
   }
   return minValue;
}
```

Now we can call this method using expressions such as `min(4,3)`, which uses two arguments, and `min(7,6,5,11)`, which uses four arguments.

The following version uses the for each loop

```
public int min(int ... args)
{
   int minValue = Integer.MAX_VALUE;
   for (int x : args)
   {
      if (x < minValue) minValue = x;
   }
   return minValue;
}
```

Here `x` takes on the values `args[0]`, `args[1]`, and so on. If we had wanted to return the position at which the first value of the minimum was found then we would need the index so the original version of the for loop would be better. ■

8.9 Two-dimensional arrays

So far we have considered only one-dimensional arrays. Two-dimensional arrays are also common. They correspond to doubly subscripted variables that occur, for example, in matrices (see Example 8.4). In mathematics the row and column indices normally start at 1 but we must take into account that indices always start at 0 in Java. As for one-dimensional arrays, a two-dimensional array can be initialized using assignment statements or an array initializer.

■ EXAMPLE 8.35 **(Defining a 2×2 matrix with assignment statements)** The BeanShell statements

```
bsh % double[][] a = new double[2][2]; // construct 2 by 2 array
bsh % a[0][0] = 1.0; // row 0, column 0
bsh % a[0][1] = 2.0; // row 0, column 1
bsh % a[1][0] = 3.0; // row 1, column 0
bsh % a[1][1] = 4.0; // row 1, column 1
```

declare a 2×2 matrix `a` in Java and initialize it to the matrix $\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$. Here the first subscript is called the row index and the second is called the column index. BeanShell does not know how to print a 2-dimensional array but it can print the rows if you use `print(a[0])` for row 0 and `print(a[1])` for row 1. ■

■ EXAMPLE 8.36 **(Defining a 2×2 matrix with an initializer)** The statement

```
double[][] a = { {1.0, 2.0}, {3.0, 4.0} };
```

declares the matrix in the preceding example using an array initializer (Java or BeanShell). This is shorthand accepted by the compiler for

```
double[][] a = new double[][] { {1.0, 2.0}, {3.0, 4.0} };
```

The initializer specifies the matrix one row at a time and it shows that a two-dimensional array is really an array of row arrays. For example, `a[0]` is a reference to row 0, with column entries `a[0][0]` and `a[0][1]`; `a[1]` is a reference to row 1, with column entries `a[1][0]` and `a[1][1]`. Therefore the matrix is a one-dimensional array of rows. ■

■ EXAMPLE 8.37 **(One-dimensional array of rows)** We can generalize and think of a two-dimensional array as a one-dimensional array of rows where each row is a one-dimensional array of elements. In fact, each row can have a different number of elements in it. An array with unequal row lengths is sometimes called a **ragged array**. If all rows have the same length the array is called a **rectangular matrix**. If the number of rows and columns are the same the array is called a **square matrix**.

For example, the statement

```
int[][] a = { {1}, {3,4,5}, {6,7,8,9} };
```

defines a two-dimensional array such that row 0 has one element, row 1 has three elements, and row 2 has four elements. The number of rows is three and is given by `a.length`. Row 0 is `a[0]` and its length is `a[0].length` (1 in this case), row 1 is `a[1]` and its length is `a[1].length` (3 in this case), and finally row 2 is `a[2]` and its length is `a[2].length` (4 in this case). Given a two-dimensional array `a` the loop

```
for (int r = 0; r < a.length; r++)
{
   System.out.println("Row " + r + " has " + a[r].length + " elements");
}
```

displays the number of elements in each row using `a.length` as the number of rows and using `a[r].length` as the length of row `r`.

Try it in BeanShell using “Capture System in/out/err”.

```
bsh % int[][] a = { {1}, {3,4,5}, {6,7,8,9} };
bsh % for(int r = 0; r < a.length; r++)
{
   System.out.println("Row " + r + " has " + a[r].length + " elements");
}
Row 0 has 1 elements
Row 1 has 3 elements
Row 2 has 4 elements
```

■

■ EXAMPLE 8.38 **(One-dimensional array of rows)** The array in Example 8.37 can also be constructed using

```
int[][] a = new int[3][]; // array of references to three rows
a[0] = new int[] {1};        // construct row 0
a[1] = new int[] {3,4,5};    // construct row 1
a[2] = new int[] {6,7,8,9};  // construct row 2
```

which clearly shows that it is an array with three rows of various lengths. The statements

```
int[][] a = new int[3][];
a[0] = new int[1];
a[1] = new int[3];
a[2] = new int[4];
```

construct this array but do not assign any values to the array elements.

There are four one-dimensional arrays here (`new` is used four times) as shown in Figure 8.10. The first is an array of three references to the rows (vertical array in the figure). The elements of

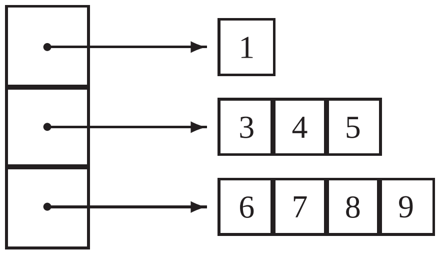


Figure 8.10: A two-dimensional array with different length rows

this array are references to rows containing 1, 3, and 4 elements respectively.

The same picture can be used to represent elements of object type. In this case where the integers appear in the picture there would instead be arrows denoting references to the objects. ■

8.9.1 Multiplying matrices

As an example of two-dimensional array manipulation let us develop a class that multiplies an $m \times p$ matrix A by an $p \times n$ matrix B to get the $m \times n$ matrix $C = AB$ called the product matrix. Recall that the matrix multiplication AB is defined only if the number of columns of A is the same as the number of rows of B and we have denoted this number by p.

The formulas for the matrix elements of C are given in Example 8.4 with indices beginning at 1. With indices beginning at zero the matrix elements c_{ij} are given by

$$c_{ij} = a_{i0}b_{0j} + \cdots + a_{i,p-1}b_{p-1,j} = \sum_{k=0}^{p-1} a_{ik}b_{kj}, \quad 0 \le i \le m-1, \quad 0 \le j \le n-1.$$

We need three nested for-loops to calculate the matrix elements c_{ij}. The outer loop index i goes from 0 to $m-1$ where m is the number of rows in A. The next loop index j goes from 0 to $n-1$ where n is the number of columns in B. This gives the loop structure

```
int m = a.length; // rows in A
int n = b[0].length; // columns in B
for (int i = 0; i <= m-1; i++)
{
   for (int j = 0; j <= n-1; j++)
   {
      // calculate c[i][j] here
   }
}
```

We have used `a.length` to get the number of rows in *A* and `b[0].length` to get the number of columns in *B*. This works since *b* is rectangular so `b[k].length` is the same for all row indices `k`.

To calculate `c[i][j]` we need to use another loop to compute the sum of *p* terms:

```
double sum = 0.0;
int p = b.length; // number of rows in B
for (int k = 0; k <= p-1; k++)
{
   sum = sum + a[i][k] * b[k][j];
}
c[i][j] = sum;
```

Putting this loop inside the outer two loops gives the following triply-nested loop structure for matrix multiplication:

```
int m = a.length; // rows in A
int n = b[0].length; // columns in B
int p = b.length; // number of rows in B
double[][] c = new double[m][n]; // create product matrix
for (int i = 0; i <= m-1; i++)
{
   for (int j = 0; j <= n-1; j++)
   {
      double sum = 0.0;
      for (int k = 0; k <= p-1; k++)
      {
         sum = sum + a[i][k] * b[k][j];
      }
      c[i][j] = sum;
   }
}
```

A method to do this matrix multiplication would have the prototype

```
public double[][] multiply(double[][] a, double[][] b)
```

indicating that the arguments are matrices and the return value is a matrix. Here is a class containing the method which can be tested using BlueJ or BeanShell.

Class `MatrixMultiplier`

book-projects/chapter8/two_d_arrays

```
package chapter8.two_d_arrays; // remove this line if you're not using packages
/**
 * Class to illustrate matrix multiplication
 */
public class MatrixMultiplier
{
   public double[][] multiply(double[][] a, double[][] b)
   {
      int m = a.length; // rows in A
      int n = b[0].length; // columns in B
      int p = b.length; // number of rows in B
      double[][] c = new double[m][n]; // create product matrix

      for (int i = 0; i <= m-1; i++)
      {
         for (int j = 0; j <= n-1; j++)
         {
            double sum = 0.0;
            for (int k = 0; k <= p-1; k++)
            {
               sum = sum + a[i][k] * b[k][j];
            }
            c[i][j] = sum;
         }
      }
      return c;
   }
}
```

Testing the method in BlueJ

It is interesting to test the `multiply` method in BlueJ. As a simple test let us compute the matrix product

$$C = AB = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 4 & 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 23 & 23 & 23 & 23 \\ 53 & 56 & 59 & 62 \end{bmatrix}$$

Perform the following steps:

1. Create a `MatrixMultiplier` object called `multiplier`.
2. From its object menu select the `multiply` method.
3. In the input box for matrix `a` enter `{{1,2,3},{4,5,6}}`.
4. In the input box for matrix `b` enter `{{1,2,3,4},{5,6,7,8},{4,3,2,1}}` and click "OK".
5. This gives the "Method Result" window showing `<object-reference>` for the return value of the method which is the product matrix. Select the object reference, click the "Get" button and give the object the name `c`. You will now have an array object on the workbench for the product matrix.

6. Right click on this object and choose “Inspect”. The resulting inspector window shows the information

   ```
   int length = 2
   [0] = <object-reference>
   [1] = <object-reference>
   ```

   indicating that this object is an array of length 2 whose rows are also object references.

7. Click on the first object reference, select “Get”, and give this object the name `row0`. It will appear on the work bench and is the first row of the product matrix.

8. Similarly click on the second object reference, select “Get”, and give this object the name `row1`. It will appear on the work bench and is the second row of the product matrix.

9. Right click on the `row0` object and select “Inspect”. The resulting inspector window shows the information

   ```
   int length = 4
   [0] = 23
   [1] = 23
   [2] = 23
   [3] = 23
   ```

   indicating that this object is an array of length 4 whose values are the first row of the product matrix.

10. Similarly, right click on the `row1` object and select “Inspect”. The resulting inspector window shows the information

    ```
    int length = 4
    [0] = 53
    [1] = 56
    [2] = 59
    [3] = 62
    ```

    indicating that this object is an array of length 4 whose values are the second row of the product matrix.

Testing the method in BeanShell

The following example shows another way to test the multiply method.

■ EXAMPLE 8.39 **(Matrix multiplication using BeanShell)** Try the following statements

```
bsh % addClassPath("c:/book-projects/chapter8/two_d_arrays");
bsh % MatrixMultiplier mult = new MatrixMultiplier();
bsh % double[][] a = {{1,2,3},{4,5,6}};
bsh % double[][] b = {{1,2,3,4},{5,6,7,8},{4,3,2,1}};
```

```
bsh % double[][] c = mult.multiply(a,b);
bsh % print(c.length);
2
bsh % print(c[0]);
double[]: {23.0,23.0,23.0,23.0,}
bsh % print(c[1]);
double[]: {53.0,56.0,59.0,62.0,}
```

to test the matrix multiplication method. ■

Testing matrix multiplication

The following runner class can be used to test the matrix multiplication method.

Class `MatrixMultiplierRunner`

book-projects/chapter8/two_d_arrays

```
package chapter8.two_d_arrays; // remove this line if you're not using packages
import java.util.Scanner;
/**
 * Class to test matrix multiplication.
 * Compute matrix product C = AB where A is an m by p
 * matrix, B is a p by n matrix and C is an m by n matrix.
 */
public class MatrixMultiplierRunner
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);

      // get matrix dimensions for  A, B

      System.out.println("Enter number of rows in A");
      int m = input.nextInt();

      System.out.println("Enter number of columns in A)");
      System.out.println("which is also the number of rows in B");
      int p = input.nextInt();

      System.out.println("Enter number of columns in B)");
      int n = input.nextInt();

      double[][] a = new double[m][p];
      double[][] b = new double[p][n];

      // Get matrix elements of A one row at a time

      System.out.println("Enter rows of A");
      for (int i = 0; i < m; i++)
      {
         for (int j = 0; j < p; j++)
         {
            a[i][j] = input.nextInt();
         }
      }

      // Get matrix elements of B one row at a time

      System.out.println("Enter rows of B");
      for (int i = 0; i < p; i++)
      {
         for (int j = 0; j < n; j++)
         {
            b[i][j] = input.nextInt();
         }
      }

     // Compute product matrix C and display all matrices

     MatrixMultiplier mult = new MatrixMultiplier();
     double[][] c = mult.multiply(a, b);
     System.out.println("Matrix a:");
     displayArray(a);
     System.out.println("Matrix b:");
     displayArray(b);
     System.out.println("Matrix c:");
     displayArray(c);
   }

   // Display a 2d-array one row at a time
   private static void displayArray(double[][] a)
   {
      for (int row = 0; row < a.length; row++)
      {
         System.out.print("Row " + row + ": ");
         for (int col = 0; col < a[0].length; col++)
         {
            System.out.print(a[row][col] + " ");
         }
         System.out.println();
      }
   }
}
```

and here is some sample output.

```
Enter number of rows in A
2
Enter number of columns in A)
which is also the number of rows in B
3
Enter number of columns in B)
```

```
4
Enter rows of A
1 2 3
4 5 6
Enter rows of B
1 2 3 4
5 6 7 8
4 3 2 1
Matrix a:
Row 0: 1.0 2.0 3.0
Row 1: 4.0 5.0 6.0
Matrix b:
Row 0: 1.0 2.0 3.0 4.0
Row 1: 5.0 6.0 7.0 8.0
Row 2: 4.0 3.0 2.0 1.0
Matrix c:
Row 0: 23.0 23.0 23.0 23.0
Row 1: 53.0 56.0 59.0 62.0
```

8.9.2 Board games

Board games provide another example of the usefulness of two-dimensional arrays. The board is a square or rectangular array of squares such as a chess board or a tic-tac-toe board. Each square can be unoccupied, or it can contain a game piece (chess piece for example, or an X or O in tic-tac-toe) for one player or the other.

As an example consider the three by three board for a game of tic-tac-toe. We can use a matrix of type `int[][]` to represent the board. A matrix element value of zero indicates that the corresponding square is unoccupied, a value of one indicates that an X is in the square and a value of two indicates that an O is in the square. We can use a two-dimensional array of the form

```
int[][] board = new int[3][3];
```

Then, the top row of squares is `board[0][k]`, for `k = 0,1,2`, the middle row is `board[1][k]`, for `k=0,1,2`, and the bottom row is `board[2][k]`, for `k=0,1,2`.

To initialize the board before starting a new game the following nested loop can be used.

```
for (int row = 0; row < board.length; row++)
{
   for (int column = 0; column < board.length; column++)
   {
      board[row][column] = 0;
   }
}
```

There are many methods that can be written which would be useful in developing a complete program for the game. For example, when a user selects a square to mark with an X or an O the program should check to see if the attempted move is legal, a move being legal only if the chosen square is unoccupied. Thus, we could write the following method.

```
public boolean isLegalMove(int row, int column)
{
   return board[row][column] == 0;
}
```

which returns `true` only if the chosen square is unoccupied.

Another useful method would be one that determines whether the board is full, indicating the game is a draw if there is no winner:

```
public boolean isBoardFull()
{
   for (int row = 0; row < board.length; row++)
   {
      for (int column = 0; column < board.length; column++)
      {
         if (board[row][column] == 0) return false; // found empty square
      }
   }
   return true; // didn’t find any empty squares
}
```

This method returns `false` if there is at least one unoccupied square on the board.

Another important method would be one with prototype

```
public boolean isWinner(int player)
```

such that `isWinner(1)` has the value `true` if the player using X won the game and `isWinner(2)` has the value `true` if the player using Y won the game (see Exercise 8.5).

8.10 Card shuffling and dealing application

An interesting application of arrays is to represent playing cards and decks of cards as objects that can be used in a card game. We first show how to represent a card as an object from a `Card` class and then we show how to represent a deck of playing cards as an object from a `CardDeck` class using an array of `Card` objects.

We consider a standard deck of 52 cards arranged into four suits (Clubs, Diamonds, Hearts, and Spades) each containing 13 cards (Ace, Two, ..., Jack, Queen, King) as shown in Figure 8.2. Each row gives the 13 cards in a suit. The numbers 0 to 3 at the right give the suit number. The numbers 0 to 12 at the top give the rank number of a card within a suit, and the numbers 0 to 51 at the left give the card index that will correspond to the index in the array representation of a card deck.

8.10.1 `Card` class

According to Table 8.2 a card can be represented either by its index in the range 0 to 51 or by a pair of numbers $(rank, suit)$ where $0 \le rank \le 12$ and $0 \le suit \le 3$. To go from one representation to the other we can use modular arithmetic as follows

	0	1	2	3	4	5	6	7	8	9	10	11	12	
0 to 12	A♣	2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣	Q♣	K♣	0
13 to 25	A♢	2♢	3♢	4♢	5♢	6♢	7♢	8♢	9♢	10♢	J♢	Q♢	K♢	1
26 to 38	A♡	2♡	3♡	4♡	5♡	6♡	7♡	8♡	9♡	10♡	J♡	Q♡	K♡	2
39 to 51	A♠	2♠	3♠	4♠	5♠	6♠	7♠	8♠	9♠	10♠	J♠	Q♠	K♠	3

Table 8.2: A deck of playing cards

- Given *index* in the range $0 \le index \le 51$, the rank and suit can be computed using the formulas $rank = index \ \mathbf{mod}\ 13$ and $suit = index \ \mathbf{div}\ 13$.
- Given *rank* and *suit* then *index* is given by $index = 13 \times suit + rank$.

For example, 2♠ has $index = 40$ so $rank = 40 \ \mathbf{mod}\ 13 = 1$ and $suit = 40 \ \mathbf{div}\ 13 = 3$. Conversely, given the rank and suit then $index = 13 \times 3 + 1 = 40$.

`Card` class design

For the `Card` class we choose the specification

```
public class Card
{
   public Card(int rank, int suit) {...}
   public Card(int index) {...}
   public String getRankName() {...}
   public String getSuitName() {...}
   public String getCardName() {...}
   public String toString() {...}
}
```

Here we have two constructors, one for each of the two ways to represent a card. Also there are "get methods" to return various string representations. The `getRankName` method will return a string such as `"Ace"` or `"King"`, the `getSuitName` method will return a string such as `"Clubs"` or `"Spades"`, the `getCardName` method returns a compact string representation such as `"A-C"` for A♣ or `"K-S"` for K♠, and the standard `toString` method returns the same string as `getCardName`.

`Card` class implementation

We now have enough information to write the class:

Class `Card`

book-projects/chapter8/card_deck

```
package chapter8.card_deck; // remove this line if you're not using packages
/**
```

```
 *  A Card object encapsulates a standard playing card in terms
 *  of 4 suits and 13 ranks. Each card can also be described in
 *  terms of an index in the range 0 to 51:
 *  <pre>
 *  index  0 to 12:   A-C, 2-C, 3-C, ..., K-C
 *  index 13 to 25:   A-D, 2-D, 3-D, ..., K-D
 *  index 26 to 38:   A-H, 2-H, 3-H, ..., K-H
 *  index 39 to 51:   A-S, 2-S, 3-S, ..., K-S</pre>
 *  The index, rank (0-12) and suit (0-3) are related by
 *  <pre>
 *  index = 13*suit + rank
 *  suit = index / 13
 *  rank = index % 13</pre>
 */
public class Card
{
   private static String[] ranks
      = { "Ace", "Two", "Three", "Four", "Five", "Six", "Seven",
         "Eight", "Nine", "Ten", "Jack", "Queen", "King" };

   private static String rankLetter = "A23456789TJQK";

   private static String[] suits
      = { "Clubs", "Diamonds", "Hearts", "Spades" };

   private static String suitLetter = "CDHS";

   private int rank;  // 0 to 12
   private int suit;  // 0 to 3
   private int index; // 0 to 51

   private String rankName; // "Ace", "Two", ... "King"
   private String suitName; // "Clubs" ... "Spades"
   private String cardName; // e.g. A-C for ace of clubs

   /**
    * Construct a card given its rank and suit.
    * @param rank card rank in range 0 (Ace) to 12 (King)
    * @param suit card suit in range 0 (Clubs} to 3 (Spades)
    */
   public Card(int rank , int suit)
   {
      this.rank = rank;
      this.suit = suit;
      index = 13*suit + rank;
      rankName = ranks[rank];
      suitName = suits[suit];
      cardName = rankLetter.substring(rank,rank+1) + "-"
         + suitLetter.substring(suit,suit+1);
   }

   /**
```

```
    * Construct a card given its index.
    * @param index card index in the range 0 (A-C) to 51 (K-S)
    */
   public Card(int index)
   {
      this.index = index;
      rank = this.index % 13;
      suit = this.index / 13;
      rankName = ranks[rank];
      suitName = suits[suit];
      cardName = rankLetter.substring(rank,rank+1) + "-"
         + suitLetter.substring(suit,suit+1);
   }

   /**
    * Return the rank name of the card.
    * @return rank name of card: "Ace" to "King"
    */
   public String getRankName()
   {
      return rankName;
   }

   /**
    * Return the suit name of the card.
    * @return suit name if card: "Clubs" to "Spades"
    */
   public String getSuitName()
   {
      return suitName;
   }

   /**
    * Return the card name.
    * @return the card name in format A-C to K-S
    */
   public String getCardName()
   {
      return cardName;
   }

   /**
    * Return a string representation of a card.
    * @return a string representation of a card.
    */
   public String toString()
   {
      return cardName;
   }
}
```

Here we use two constant arrays for the names of the ranks and the names of the suits. The numeric rank and suit values are used as indices into these arrays to determine `rankName` and `suitName`.

Similarly two strings are used to obtain single letter representations (using substring) of the rank and suit which are used to determine `cardName`.

This class can now be tested either with BlueJ or with BeanShell. The following example shows how to test it with BeanShell using Table 8.2 to check results.

■ EXAMPLE 8.40 **(Testing the `Card` class)** Try statements such as

```
bsh % addClassPath("c:/book-projects/chapter8/card_deck");
bsh % Card c = new Card(35);
bsh % print(c.getCardName());
T-H
bsh % print(c.getRankName());
Ten
bsh % print(c.getSuitName());
Hearts
bsh % print(c);
T-H
```

to test the `Card` class. ■

8.10.2 `CardDeck` class

A `CardDeck` object represents a deck of 52 `Card` objects.

`CardDeck` class design

The most fundamental operations for a deck of cards is to put the deck in some standard order (done by the constructor), shuffle the deck, and deal a card. This gives the following specification.

```
public class CardDeck
{
   public CardDeck() {...}
   public void shuffle() {...}
   public Card deal() {...}
   public int cardsInDeck() {...}
   public boolean empty() {...}
   public String toString() {...}
}
```

Here the constructor creates a deck in the standard order given in Table 8.2, the `shuffle` method shuffles the deck into some random order, the `deal` method returns the top card and removes it from the deck, the `cardsInDeck` method returns a number in the range 0 to 52 indicating how many cards remain to be dealt, the `empty` method returns true if there are no more cards in the deck, and the `toString` method returns a string representation of the cards remaining in the deck.

`CardDeck` implementation

To implement the class we need an array of `Card` objects as a private data field and an index that keeps track of the array index of the current top card:

```
private static final int DECK_SIZE = 52;
private Card[] deck;
private int topCardIndex;
```

Here for a full deck `topCardIndex` will be zero. Each time a card is dealt this index is incremented to refer to the new top card. When there are no cards remaining the index has the value 52 which is one more than the largest array index.

The constructor implementation

```
public CardDeck()
{
   deck = new Card[DECK_SIZE];
   initialize();
}
```

constructs the deck and calls a private `initialize` method which constructs 52 cards in standard order:

```
private void initialize()
{
   topCardIndex = 0;
   for (int k = 0; k < DECK_SIZE; k++)
   {
      deck[k] = new Card(k);
   }
}
```

The only tricky method is `shuffle`. We use the `Random` class in `java.util` to generate random integers:

```
Random rand = new Random();
```

Then `rand.nextInt(n)` returns a random integer in the range 0 to `n-1`. The following algorithm can be used to shuffle the 52 cards.

- Step 0: Choose a random position from 0 to 51 and exchange the element at that index with the element at position 0. This gives a random card in position 0.

- Step 1: Choose a random position from 1 to 51 and exchange the element at that index with the element at position 1. This gives a random card in position 1.

- Step k: Choose a random position from k to 51 and exchange the element at that index with the element at position k. This gives a random card in position k.

- Step 50: Choose a random position from 50 to 51 and exchange the element at that index with the element at position 50. This gives a random card in position 50. This is the last step since the last card at position 51 will be random.

We can generate a random integer index in the range k to 51 using

```
int index = rand.nextInt(DECK_SIZE - k) + k;
```

The `rand` operation here produces a random integer in the range 0 to $52-k-1$ and we add k to get a random integer in the range k to 51. This gives the following implementation of `shuffle`.

```
public void shuffle()
{
   Random rand = new Random();
   initialize();
   for (int k = 0; k <= DECK_SIZE - 2; k++)
   {
      int index = rand.nextInt(DECK_SIZE - k) + k;
      Card temp = deck[k];
      deck[k] = deck[index];
      deck[index] = temp;
   }
}
```

To deal a card from the deck we need to return a reference to the `Card` object whose index is `topCardIndex` and then we need to increment `topCardIndex` so that it references the next card. If the `deal` method is called on an empty deck of cards it returns `null`:

```
public Card deal()
{
   if (topCardIndex == DECK_SIZE) return null;
   Card topCard = deck[topCardIndex];
   topCardIndex++;
   return topCard;
}
```

The other methods are easily implemented and we obtain the class

Class `CardDeck`

book-projects/chapter8/card_deck

```
package chapter8.card_deck; // remove this line if you’re not using packages
import java.util.Random;
/**
 * A CardDeck object represents a deck of 52 Card objects.
 */
public class CardDeck
{
```

```
private static final int DECK_SIZE = 52;
private Card[] deck;
private int topCardIndex;

/**
 * Construct a deck of cards initialized in the standard order 0 to 51
 */
public CardDeck()
{
   deck = new Card[DECK_SIZE];
   initialize();
}

/**
 * Return all the cards to the deck in standard order.
 */
public void initialize()
{
   topCardIndex = 0;
   for (int k = 0; k < DECK_SIZE; k++)
   {
      deck[k] = new Card(k);
   }
}

/**
 * Return all the cards in the deck to standard order and
 * then shuffle them into a random order.
 */
public void shuffle()
{
   Random rand = new Random();
   initialize();
   for (int k = 0; k <= DECK_SIZE - 2; k++)
   {
      int index = rand.nextInt(DECK_SIZE - k) + k;
      Card temp = deck[k];
      deck[k] = deck[index];
      deck[index] = temp;
   }
}

/**
 * Deal a card from the deck.
 * If there are no more cards null is returned.
 * @return the card or null if there are no more cards.
 */
public Card deal()
{
   if (topCardIndex == DECK_SIZE) return null;
   Card topCard = deck[topCardIndex];
   topCardIndex++;
```

```
      return topCard;
   }

   /**
    * Return the number of cards remaining in the deck.
    * @return the number of cards remaining in the deck.
    */
   public int cardsInDeck()
   {
      return DECK_SIZE - topCardIndex;
   }

   /**
    * Return true if deck is empty, false otherwise.
    * @return true if deck is empty, false otherwise.
    */
   public boolean empty()
   {
      return topCardIndex == DECK_SIZE;
   }

   /**
    * Return a string representation of the cards remaining in the deck.
    */
   public String toString()
   {
      StringBuilder b = new StringBuilder(215);
      for (int k = topCardIndex; k < DECK_SIZE; k++)
      {
         Card card = deck[k];
         b.append(card.getCardName());
         if ( (k+1) % 13 == 0)
            b.append("\n");
         else
            b.append(" ");
      }
      return b.toString();
   }
}
```

The `toString` method uses the `StringBuilder` class introduced in Section 7.10.2 to produce a string representation of the cards remaining in the deck, 13 cards per line.

Testing the class in BlueJ

Try steps such as the following to test the class in BlueJ.

1. Construct a `CardDeck` object.

2. From its object menu select `shuffle` then use the `toString` method to see the first few cards of the deck (the newlines in the string are replaced by spaces in BlueJ).

3. From the object menu select the `deal` method. From the “Method Result” window select the object reference and use “Get” to put the `Card` object on the workbench.

4. Use the `Card` object’s methods to see the card’s properties.

5. From the `CardDeck` object menu select the `cardsInDeck` method to verify that there are now 51 cards in the deck.

Testing the class in BeanShell

■ EXAMPLE 8.41 **(Testing the `CardDeck` class)** Try statements such as

```
bsh % addClassPath("c:/book-projects/chapter8/card_deck");
bsh % CardDeck deck = new CardDeck();
bsh % deck.shuffle();
bsh % print(deck);
T-S 7-S A-S Q-D 5-C 9-H 7-C 5-D 6-C 4-H J-C 5-S T-H
9-D K-H J-S 4-D K-D 3-C 3-H K-S T-D 8-D Q-H A-C 3-S
J-D A-D 8-H 9-S 7-H 4-S 5-H 8-C 2-H K-C Q-S T-C 6-D
8-S 3-D 7-D J-H 6-H 6-S 4-C 2-S 2-C 9-C 2-D Q-C A-H

bsh % Card top = deck.deal();
bsh % print(top);
T-S
bsh % print(deck.cardsInDeck());
51
```

to test the `CardDeck` class using BeanShell. ■

Testing the class from the command line

The following simple class can be used to test the class from the command line.

Class `CardDeckTester`

book-projects/chapter8/card_deck

```
package chapter8.card_deck; // remove this line if you’re not using packages
/**
 * A simple test class for CardDeck class
 */
public class CardDeckTester
{
   public void run()
   {
      CardDeck deck = new CardDeck();
      System.out.println("Initilized deck:");
      System.out.println(deck);
      System.out.println("Shuffled deck:");
      deck.shuffle();
      System.out.println(deck);

      // Deal a few cards

      Card card;
      card = deck.deal();
      System.out.println("dealing " + card.getCardName());
      card = deck.deal();
      System.out.println("dealing " + card.getCardName());

      int count = 0;
      while ( ! deck.empty() )
      {
         Card c = deck.deal();
         count++;
         System.out.print(c.getCardName());
         if (count == 13)
         {
            System.out.println();
            count = 0;
         }
         else
         {
            System.out.print(" ");
         }
      }
      if (count != 0) System.out.println();
   }

   public static void main(String[] args)
   {
      new CardDeckTester().run();
   }
}
```

A while-loop is used to deal the cards and display them as they are dealt, 13 cards per line. Here is some typical output.

```
Initilized deck:
A-C 2-C 3-C 4-C 5-C 6-C 7-C 8-C 9-C T-C J-C Q-C K-C
A-D 2-D 3-D 4-D 5-D 6-D 7-D 8-D 9-D T-D J-D Q-D K-D
A-H 2-H 3-H 4-H 5-H 6-H 7-H 8-H 9-H T-H J-H Q-H K-H
A-S 2-S 3-S 4-S 5-S 6-S 7-S 8-S 9-S T-S J-S Q-S K-S

Shuffled deck:
9-S 7-D J-S 4-S 3-S 8-H 7-S T-D 4-C 9-C Q-C 6-H 2-S
5-C 2-C 8-C J-C 3-H 6-C Q-D Q-H 8-S 5-H 8-D K-D 3-C
T-C Q-S T-S J-D A-S 9-H 4-D 4-H K-C T-H 2-H 9-D A-C
5-S 2-D J-H A-D 7-C K-H 5-D K-S 3-D 7-H 6-S A-H 6-D

dealing 9-S
dealing 7-D
```

```
J-S 4-S 3-S 8-H 7-S T-D 4-C 9-C Q-C 6-H 2-S 5-C 2-C
8-C J-C 3-H 6-C Q-D Q-H 8-S 5-H 8-D K-D 3-C T-C Q-S
T-S J-D A-S 9-H 4-D 4-H K-C T-H 2-H 9-D A-C 5-S 2-D
J-H A-D 7-C K-H 5-D K-S 3-D 7-H 6-S A-H 6-D
```

8.11 Review exercises

▶ **Review Exercise 8.1** Draw pictures of an array of three `BankAccount` objects that shows the three steps in the creation process.

▶ **Review Exercise 8.2** Draw a picture that shows that a 2-dimensional array is an array of rows.

▶ **Review Exercise 8.3** Write a pseudo-code algorithm that adds corresponding elements of the given arrays $\langle a_0, \ldots, a_{n-1} \rangle$ and $\langle b_0, \ldots, b_{n-1} \rangle$ to produce the array $\langle c_0, \ldots, c_{n-1} \rangle$, using the addition formula $c_k = a_k + b_k$.

▶ **Review Exercise 8.4** Write a pseudo-code algorithm called `max` that finds the maximum element in an $m \times n$ rectangular array $\langle a_{00}, \ldots, a_{mn} \rangle$ of numbers.

8.12 BeanShell exercises

The following BeanShell exercises can be done using the Workspace Editor. First run BeanShell, then choose “Workspace Editor” from the “File” menu to open the editor. If you want to use `System.out.println` then it is also necessary to choose “Capture System in/out/err” from the “File” menu.

Now you can type statements into the editor and they won’t be executed as they are entered. When you have finished entering statements choose “Evaluate in Workspace” from the “Evaluate” menu. Now the statements will be executed. You can edit the statements and evaluate them again, and so on.

This is useful for testing static methods. Type in the method, evaluate it then test it interactively using the workspace.

▶ **BeanShell Exercise 8.1** Write some statements to calculate 2^0 to 2^{15} in a for-loop and store them in an array called `powerOfTwo` such that 2^k is stored in `powerOfTwo[k]`.

▶ **BeanShell Exercise 8.2** Write some statements to calculate $n!$ for $n = 0$ to $n = 12$ in a for-loop and store them in an array called `fact` such that $n!$ is stored in `fact[n]`.

▶ **BeanShell Exercise 8.3** Write some statements to create an array of 3 `BankAccount` objects and write a for-loop to compute the total of their balances.

▶ **BeanShell Exercise 8.4** Translate the algorithm in Review Exercise 8.3 to a method with prototype

```
public double[] add(double[] a, double[] b)
```

The method should construct the array for the sum and return a reference to it. Test the method using BeanShell.

► **BeanShell Exercise 8.5** Write a Java method called `max` with prototype

```
public double max(double[][] a)
```

that finds the maximum element in an $m \times n$ rectangular array $\langle a_{00}, \ldots, a_{mn} \rangle$ of numbers. Test the method using BeanShell.

► **BeanShell Exercise 8.6** Do a more general version of the `max` method in BeanShell Exercise 8.5 that does not assume that array is rectangular. In other words each row can have a different number of elements. Recall that the number of rows in a 2-dimensional matrix `a` is `a.length` and the number of elements in row `k` is `a[k].length`.

► **BeanShell Exercise 8.7** Write a method with prototype

```
public boolean same(double[][] a)
```

that returns true if all the elements in the array `a` are the same and false otherwise.

► **BeanShell Exercise 8.8** Write a method with prototype

```
public Polynomial add(Polynomial p1, Polynomial p2}
```

that adds two `Polynomial` objects and returns the sum as a `Polynomial` object. (If polynomial p_1 has degree m and polynomial p_2 has degree n, then their sum has degree $s = \max(m,n)$)

8.13 Programming exercises

► **Exercise 8.1 (Reversing the elements of an array)**
Write a method with prototype

```
public void reverse(int[] a)
```

that reverses the elements of the array `a` in place (without creating another array). Write a class to test the method using the `IntArrayIO` class for input and output. Hint: swap elements from the end with elements from the beginning in a for-loop.

► **Exercise 8.2 (A student mark histogram)**
Write a method with prototype

```
public int[] markHistogram(int[] marks)
```

where `marks` is an array of marks in the range 0 to 100. The method creates a new array h of length 6 such that h_0 is the number of marks m with $0 \le m < 50$, h_1 is the number of marks m with $50 \le m < 59$, h_2 is the number of marks m with $60 \le m < 69$, and so on until h_5 is the number of marks m with $90 \le m \le 100$. This array is returned as the value of the method. Write a class to test the method by reading an array of marks and displaying the number of marks in each range.

► **Exercise 8.3 (Sorting bank accounts)**
Write a program called `SortBankAccounts` that

1. asks the user how many bank accounts will be entered,
2. reads the data for that many bank accounts and stores them in an array,
3. sorts the array in decreasing order by balance and displays the sorted array,
4. sorts the array in increasing lexicographical order according to the owner name and displays the sorted array.

► **Exercise 8.4 (Pascal's triangle)**
Write a program called `PascalTriangleMaker` that reads an integer value n with $n \geq 0$. computes the binomial coefficients

$$\binom{n}{k} = C(n,k) = \frac{n!}{k!(n-k)!}, \; n \geq 0, k = 0, \ldots n,$$

the number of k-element subsets of an n-element set, using the recurrence relation

$$C(n,k) = C(n-1,k) + C(n-1,k-1), \text{ with } C(n,0) = 1, C(n,n) = 1,$$

which expresses a value in row n in terms of two neighboring values in the preceding row $n-1$. Store the calculated coefficients in a two-dimensional ragged array: row n should have $n+1$ elements in it to store $C(n,0), C(n,1), \ldots, C(n.n)$. Display the array in the triangular form

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
```

How large can n be before integer overflow occurs?

► **Exercise 8.5 (An isWinner method for tic-tac-toe)**
We have represented a tic-tac-toe board by a two-dimensional array called `board`. Write a method called `isWinner` with prototype

```
public boolean isWinner(int player)
```

If the value of `player` is 1, corresponding to X, the method should return true if this player has won the game with three X's in a row, or in a column, or in one of the two diagonals. Similarly, if the value of `player` is 2, corresponding to O, the method should return true if this player has won the game.

Test your method by including it in a class called `TicTacToeTester` that gets the 9 square values, one row at a time, as command-line arguments and displays either that the player using X has won, the player using Y has won, or the game is a draw (board is full but there is no winner). For example, here is the output for three program runs.

```
java TicTacToeTester 2 2 1 2 1 0 1 0 0
Player using X is the winner
java TicTacToeTester 1 1 2 1 2 0 2 0 0
Player using O is the winner
java TicTacToeTester 2 2 1 1 1 2 2 1 2
The game is a draw
```

► Exercise 8.6 (A bar graph class)
Using the `LineGraph` class as a model write a similar `BarGraph` class for a graph of vertical bars filled with random colors and outlined in black. The bar heights are stored in an array $h = \langle h_0, h_1, \ldots, h_{n-1}\rangle$. In the world coordinate system the width of each bar is 1 unit, so the top left corner of bar i has coordinates (i, h_i).

Write a runner class called `BarGraphRunner` that uses command-line arguments to test the `BarGraph` class. For the command

```
java BarGraphRunner 1 2 3 4 5 4 3 2 1
```

A typical output window is shown in Figure 5.20.

► Exercise 8.7 (Drawing a pentagon using arrays)
Rewrite the `DrawPentagon` program from Chapter 5 using arrays of points and for-loops. An array for the five vertices of the pentagon can be declared using

```
private Point2D.Double[] v = new Point2D.Double[5];
```

► Exercise 8.8 (Drawing a pentagonal star)
Write a program called `PentagonalStar` to draw a five pointed star, similar to the one shown in Figure 8.11, as follows. Define the pentagonal angles and vertices of the inner pentagon by

$$\alpha_k = \frac{\pi}{180}(72k + 54), \;\; v_k = (r_1 \cos\alpha_k, r_1 \sin\alpha_k), \; k = 0, \ldots, 4$$

Define the pentagonal star angles and vertices by

$$\beta_k = \frac{\pi}{180}(72k + 90), \;\; w_k = (r_2 \cos\beta_k, r_2 \sin\beta_k), \; k = 0, \ldots, 4$$

The two radii, r_1 and r_2, are connected by the formula

$$r_1 = \left(\frac{\sin a}{\cos b}\right) r_2, \text{ where } a = \left(\frac{\pi}{180}\right) 18, \text{ and } b = \left(\frac{\pi}{180}\right) 36$$

The radius r_2 is the radius of the circumscribed circle for the star, and the smaller radius r_1 is the radius of the pentagon inside the star (formed from the 5 vertices of the star that are closest to the center).

Choose an appropriate coordinate system and value for r_2 and draw the pentagonal star as 10 lines between the following 10 pairs of vertices:

$$v_0w_0,\; w_0v_1,\; v_1w_1,\; w_1v_2,\; v_2w_2,\; w_2v_3,\; v_3w_3,\; w_3v_4,\; v_4w_4,\; w_4v_0$$

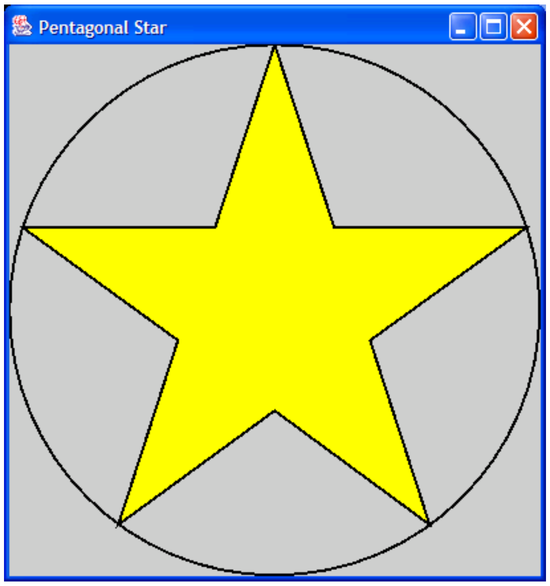


Figure 8.11: Output of the `PentagonalStar` program

► Exercise 8.9 (Drawing a duck as an array of lines)
The following class

```
public class Duck
{
   public static double minX = 0.0;
   public static double maxX = 29.0;
   public static double minY = 0.7;
   public static double maxY = 17.0;

   public static double[] x = {
       0.0,  2.0,  4.0,  6.0,  8.0, 10.0, 12.0, 14.0, 16.4, 17.0,
      17.3, 17.8, 18.5, 20.0, 22.0, 24.0, 26.0, 28.0, 29.0, 28.8,
      27.2, 25.0, 23.0, 21.5, 21.1, 21.5, 22.8, 24.1, 25.1, 25.2,
      24.2, 22.1, 20.8, 18.0, 16.0, 14.0, 12.0, 10.0,  8.0,  6.1,
       4.2,  3.0,  1.3,  0.0
   };

   public static double[] y = {
       8.8,  7.6,  7.1,  7.4,  8.0,  8.9,  9.6,  9.9,  9.4,  9.7,
      12.0, 14.0, 16.1, 17.0, 17.0, 16.0, 13.9, 13.1, 13.2, 12.3,
      11.5, 11.5, 11.5, 11.2, 10.5,  9.0,  8.0,  7.0,  5.1,  3.6,
       1.9,  1.1,  0.9,  0.7,  0.8,  0.9,  1.0,  1.2,  1.5,  2.1,
       2.9,  4.1,  6.0,  8.8
   };
}
```

defines a duck as a two arrays of *x* and *y* coordinates. Write a class called `DrawDuck` that draws the duck using a transformed coordinate system. Draw the duck as a `GeneralPath` object, outline it in black and fill it with green. The output window is shown in Figure 8.12

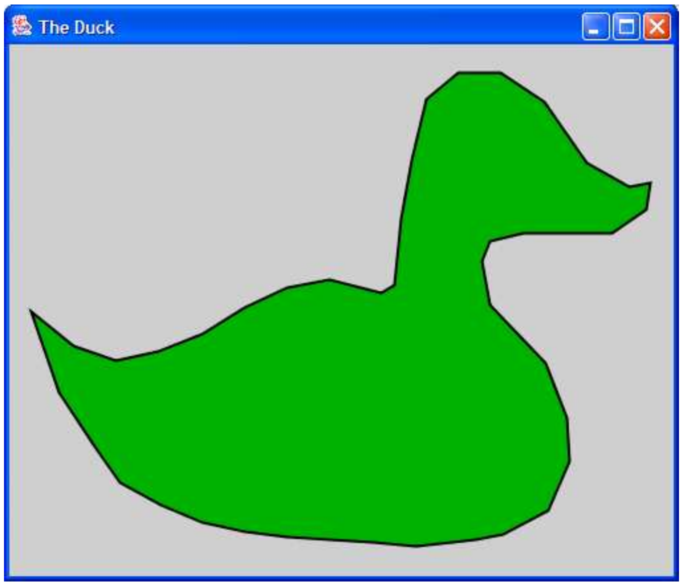


Figure 8.12: A duck using arrays of points

► **Exercise 8.10 (Returning multiple values from a method)**
If we want to find both the maximum and minimum values in an array of integers we could use the `findMinimum` and `findMaximum` methods in class `IntArrayMaxMin`. This is inefficient since both values can be found using one for-loop. We would like to write a method called `findMaxMin` that returns both values but a Java method can only return one value. To do this we invent a small class

```
public class IntRange
{
   public int min;
   public int max;
}
```

with two public data fields and no methods. Now the `findMaxMin` method can be written with prototype

```
public IntRange findMaxMin(int[] a)
```

so that it returns an `IntRange` object. The method can be called using a statement such as

```
IntRange r = findMaxMin(a);
```

Then the minimum and maximum values can be obtained as `r.min` and `r.max` since public data fields can be accessed directly using the dot notation without the need for get methods. Write this method and test it: you could put it in `IntArrayMaxMin` as a `static` method and modify `MaxMinCalculator` to test it.

► **Exercise 8.11 (Counting the number of times the maximum occurs)**
Write a pseudo-code algorithm that takes an array $\langle a_0, \ldots, a_{n-1} \rangle$ as input and returns the number of times the maximum array element occurs in the array. The obvious algorithm would use one for-loop to find the maximum value followed by another for-loop to count how many times the maximum occurs. Instead, write your algorithm using only a single for-loop, translate it into a method with prototype

```
public int countMax(double[] a)
```

and write a tester class for it.

► **Exercise 8.12** Write a class called `PolynomialAdder` that uses the `add` method from BeanShell Exercise 8.8 to add two polynomials.

► **Exercise 8.13** Write a class that can produce monthly calendars of the form

```
July 2003
 S  M  T  W  T  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 28
27 28 29 30 31
```

Internally store a calendar as a 2-dimensional array with 6 rows and 7 columns. Store 0's in positions that should be blank and store the day (1 to 31) in other positions. Use this array to print the calendar. You can use the `CalendarMonth` class in Chapter 4 (page 133) to determine on which day is the first of the month and how many days are in the month.

Chapter 9

Inheritance and Interfaces

Polymorphism for Classes and Interfaces

Outline

What is inheritance?

Rules for declaring subclasses

Simple examples of subclasses

Polymorphism

Abstract classes and polymorphism

The `Object` Class

Final classes

Interfaces

Multiple interfaces

Implementing the `Shape` interface

Turtle graphics class

Writing turtle graphics programs

Numerical applications of interfaces

9.1 Introduction

There are two hierarchies in object-oriented programming that help manage the complexity of large software systems.

The first is the object hierarchy considered in Chapter 4, that uses aggregation (composition) to construct more complex objects in terms of simpler ones, which in turn can be used to construct even more complex objects, and so on.

The second is the class hierarchy defined by inheritance. With inheritance we can define a subclass of a class (the superclass) that inherits all the functionality (methods) of the superclass. Some of these methods can be retained, or they can be overridden by providing new versions in the subclass, and new methods (functionality) can be added. This leads to a class inheritance hierarchy. The `Object` class is implicitly at the top of any inheritance hierarchy.

Java has only single-inheritance hierarchies for classes, which means that a subclass can inherit from only one superclass, but there are special important kinds of classes called interfaces, which we will also discuss, that permit a very useful form of multiple inheritance.

Abstract classes are also introduced. An abstract class declares one or more abstract methods but not their implementations and possibly other methods that do have implementations. It is up to the subclasses of an abstract class to provide implementations for the abstract methods. An `Employee` class hierarchy will be used as an example.

The related and important concept of polymorphism is also introduced. This permits all objects from classes in an inheritance hierarchy to be considered of the “same” type, which results in a uniform processing of objects in the hierarchy, without regard to the particular subclasses the objects really belong to, using the polymorphic methods of the class. The particular class that the object belongs to is not determined until run-time.

A polymorphic method is a method appearing in each subclass in an inheritance hierarchy with the same prototype. Each subclass can have its own version of this method. For an abstract class any abstract method is polymorphic. As an example we show how methods in an `Employee` hierarchy can be used to process employee salaries polymorphically.

A more general and often more important form of polymorphism is also possible using interfaces. An interface is like a special kind of abstract class that declares only method prototypes (no implementations). One can also have interface hierarchies where a subinterface inherits from a superinterface.

An important example is the `Shape` interface for defining geometrical shapes, drawing, and filling them (see Chapter 5). We will show how to make our own `Shape` objects.

Any class that implements the methods of an interface is said to implement the interface. Unlike classes interfaces also permit a useful form of multiple inheritance since it is possible for a class to implement several interfaces.

The difference compared to an inheritance hierarchy is that the set of classes that implement a particular interface do not need to be related in any other way. In particular they do not need to form a class inheritance hierarchy. Nevertheless, like classes in an inheritance hierarchy, they can be considered to be of the “same type”, namely the interface type. This leads to a more general form of polymorphism within this set of classes that implement the interface.

9.2 What is inheritance?

Inheritance defines a relationship between two classes. One is called the **superclass** and the other is called the **subclass**. Sometimes the superclass is called the parent class and the subclass is called the child class. This relationship sets up an inheritance hierarchy since a superclass can be a subclass of another class and so on until a class at the top of the hierarchy is reached that is the superclass of all lower classes. We say that the subclass inherits from the superclass. In Java the special `Object` class is at the top of all inheritance hierarchies. It is the ultimate superclass of all Java classes in the sense that an object from any class is a kind of `Object`.

We also express the superclass-subclass relationship in Java by saying that one class (the subclass) **extends** the other class (the superclass). We have already seen examples of inheritance since our classes in Chapter 5 all extend the `JPanel` class. Most of the complex functionality of displaying a window containing our graphics output was provided by `JPanel` and its superclasses and we did not need to understand how it works.

The importance of the inheritance hierarchy in the management of complex software is that each subclass can be constructed incrementally from its immediate superclass. This promotes code reuse since a subclass only specifies how its objects differ from those of the parent class. Thus, we take a given class and extend it to provide some additional functionality which can be specified in the subclass in three basic ways: (1) declare new data fields, (2) declare new methods, and (3) provide new versions of existing superclass methods (called overriding a superclass method). Of course all the public methods of the superclass that were not overridden in the subclass are automatically available in the subclass.

■ EXAMPLE 9.1 **(Domestic animal hierarchy)** We are familiar with many hierarchies such as family trees and classification systems in biology. Let us consider the part of the animal kingdom that contains domestic animals. At the top of this hierarchy we have a class called `DomesticAnimal` which specifies features common to all domestic animals. There are many subclasses. Two important ones are the `Dog` and `Cat` classes. These three classes are called abstract classes since they do not describe real animals. There are no `DomesticAnimal`, `Dog`, or `Cat` objects, only dogs or cats of particular breeds such as `Terrier` or `Persian`. Dogs and cats in each subclass have features which distinguish them from dogs and cats in other subclasses. For example, Cheshire cats are always grinning and they can make themselves invisible, unlike cats in other classes. A part of the `DomesticAnimal` hierarchy is shown in Figure 9.1 as a tree diagram. ■

■ EXAMPLE 9.2 **(Bank account inheritance hierarchy)** The `BankAccount` class used in Chapter 9 has instance data fields for an account number, owner name, and balance. Suppose we want to consider bank accounts that have a joint owner. We can extend the `BankAccount` class to obtain a subclass called `JointBankAccount`. This class provides a new instance data field for the joint owner and a new method called `getJointName` to return the joint owner name. Also, the `toString` method needs to be overridden to include the new data field. All other methods from the `BankAccount` class, such as `withdraw` and `deposit`, are automatically available in the subclass and do not need to be overridden. Extending the existing class is a lot easier than writing a completely new class from scratch that has considerable overlap with the existing `BankAccount` class. The bank account hierarchy diagram is shown in Figure 9.2 which uses a more compact way to

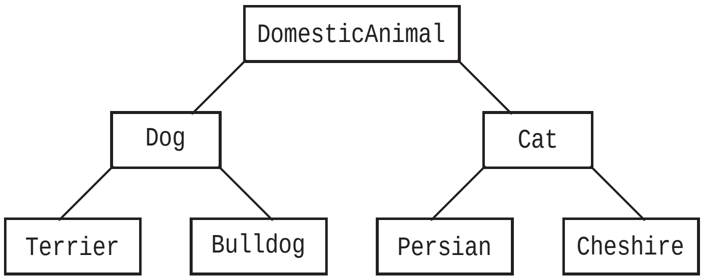


Figure 9.1: Part of the domestic animal inheritance hierarchy

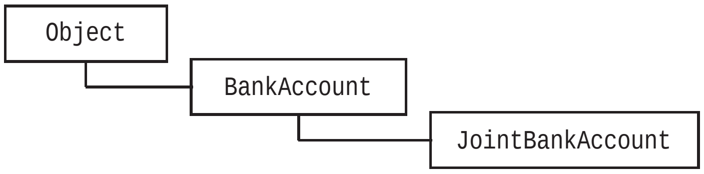


Figure 9.2: Bank account inheritance hierarchy

represent tree diagrams than Figure 9.1. The diagram shows that `JointBankAccount` is a subclass of `BankAccount` which is a subclass of `Object`. Since the `Object` class is always at the top of any hierarchy we won't normally show it on inheritance diagrams. ■

■ EXAMPLE 9.3 **(Employee inheritance hierarchy)** We can classify the various kinds of employees in a company using inheritance. At the top of the hierarchy is a class called `Employee` that represents everything common to all employees, such as name, employee number, and the date the employee was hired. This class is called the **base class**. In this example it is also a generic or abstract class. To obtain real employee classes we need to make subclasses for each kind of employee. For example, different kinds of real employees can be distinguished by the method for calculating their monthly salary: a `Manager` object has a fixed monthly salary and deductions, an `HourlyWorker` has hours worked, an hourly rate and deductions, a `PartTimeWorker` is the same but with no deductions, and a `CommissionWorker` has a fixed monthly salary and deductions, plus a commission that is a certain percentage of monthly sales. One way to design the class hierarchy is shown in Figure 9.3. ■

9.2.1 The "is-a" and "has-a" relationships

The aggregation and class inheritance hierarchies can be used to define two relationships between objects.

Inheritance is often called the "is-a" or "is-a-type-of" relationship. For example, in the bank account hierarchy a `JointBankAccount` object is a kind of `BankAccount` object. In the employee hierarchy a `Manager` object is a kind of `Employee` object. In the `DomesticAnimal` hierarchy a `Terrier` is a kind of `Dog` and a `Dog` is a kind of `DomesticAnimal`.

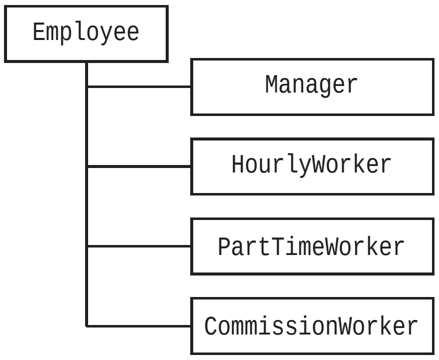


Figure 9.3: Employee inheritance hierarchy

```
public class SubclassName extends ClassName
{
    declarations for new data fields, if any
    Constructor declarations (they are never inherited)
    Method declarations for new methods, if any
    Method declarations for overridden methods, if any
}
```

Figure 9.4: A template for a simple Java subclass declaration.

Aggregation is often called the “has-a” relationship. For example, in the `Circle` class from Chapter 4, page 122, which was used to illustrate aggregation, we say that a `Circle` object “has a” `Point` object, namely the center of the circle.

Deciding which of the two hierarchies to use in a given situation can be determined by stating relationships between classes and objects in terms of the “is-a” and ”has-a” relationships. For example, it does not make sense to say that a `Circle` object “is a ” `Point` object.

9.3 Rules for declaring subclasses

In Java the keyword **`extends`** is used to denote the subclass relationship. To indicate that a class with name *SubclassName* is a subclass of *ClassName* we use a class declaration whose template is shown in Figure 9.4. The first line is like a normal class declaration except for the keyword `extends` which is followed by the name of the class to be extended (the superclass). As mentioned above, the subclass declaration only specifies how the subclass differs from the superclass according to the following basic rules (you can refer back to these rules as you read the Chapter):

1. A data field of the superclass is automatically a data field of a subclass. It is an error to declare it again in a subclass. Direct access to superclass data fields by a subclass follows the rules

   (a) A **`public`** data field can be directly accessed by any class, subclass or not.

   (b) A **`private`** data field can never be directly accessed by any other class.

   (c) A **`protected`** data field can be directly accessed by a subclass but not by any class outside the hierarchy. Protected means public for subclasses and private for other classes.

2. A subclass may declare new data fields.

3. Superclass constructors are never inherited so a subclass must provide its own constructors. In doing so, a subclass constructor may call a superclass constructor, as its first statement, using the syntax **`super(`***actualArgList***`)`** to construct the superclass part of an object.

4. A subclass may declare new methods.

5. A subclass may declare a new version of any public or protected superclass method to provide additional or new functionality not provided by the superclass method. This is called **method overriding**. In doing so, a subclass method can call the superclass version of the method using the syntax

   `super`. *methodName***`(`** *actualArgList* **`)`**.

6. Public or protected superclass methods that are not overridden by the subclass are automatically available to the subclass and are never declared again in the subclass.

9.4 Simple examples of subclasses

We now consider some simple examples to illustrate the rules.

9.4.1 Graphics programs

The simple graphics classes in Chapter 5 use inheritance. Recall that each graphics class had the structure

```
public class MyGraphicsClass extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g)
      Graphics2D g2D = (Graphics2D) g;
      // additional statements to draw on the panel
   }
```

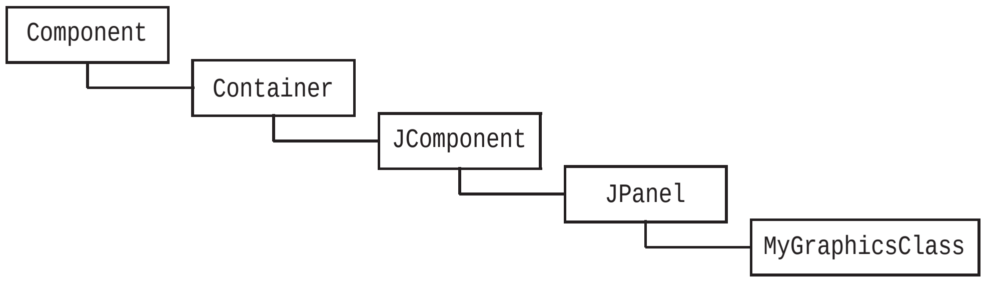


Figure 9.5: Part of the GUI component hierarchy

```
      // other methods
   }
```

Here we are extending a class called `JPanel` which is part of the class hierarchy shown in Figure 9.5. Our class, indicated by `MyGraphicsClass`, is shown at the bottom of the hierarchy. The reason that our graphics classes were fairly small is that most of the work of supporting the complex graphical user interface is done by classes higher up in the hierarchy. All we do is extend `JPanel` and override its `paintComponent` method and use

```
super.paintComponent(g)
```

at the beginning of our method to let the superclass do its share of the work. In some of our graphics classes we also used the `getWidth` and `getHeight` methods to return the width and height of the drawing panel in pixels. If you look in the `JPanel` class documentation you won't find these methods. However, you will find them higher up the hierarchy in the `JComponent` class so they are automatically inherited by our class. This is the power of inheritance. The GUI component hierarchy will be considered in more detail in Chapter 10.

9.4.2 Extending a circle calculator class

Consider the following version of the `CircleCalculator` class from Chapter 3, page 63.

Class `CircleCalculatorA`

book-project/chapter9/geometry

```
package chapter9.geometry;
/**
 * A simple class whose objects know how to calculate
 * the area of a circle given its radius.
 */
public class CircleCalculatorA
{
   protected double radius;
   private double area;
   /**
    * Construct a circle.
    * @param r the radius of the circle
    */
   public CircleCalculatorA(double r)
   {
      radius = r;
      area = Math.PI * radius * radius;
   }

   /**
    * @return the radius of the circle
    */
   public double getRadius()
   {
      return radius;
   }

   /**
    * @return the area of the circle
    */
   public double getArea()
   {
      return area;
   }
}
```

This class computes only the area of the circle. The `radius` data field declaration, which was `private`, is now `protected` so that subclasses can access it directly.

Suppose we also want to compute the circumference of the circle. Using inheritance we can create a subclass called `CircleCalculatorB` that extends `CircleCalculatorA` and also calculates the circumference. A constructor for this subclass will call the superclass constructor to compute the area, and then it will do its part and calculate the circumference. A new data field for the circumference and a corresponding inquiry method are needed.

Class `CircleCalculatorB`

book-project/chapter9/geometry

```
package chapter9.geometry;
/**
 * A class whose objects know how to calculate
 * the area and circumference of a circle given its radius.
 */
public class CircleCalculatorB extends CircleCalculatorA
{
   private double circumference; // new instance data field

   /**
    * Construct a circle
    * @param r the radius of the circle
    */
   public CircleCalculatorB(double r)
   {
      super(r);
      circumference = 2.0 * Math.PI * radius;
   }

   /**
    * @return the circumference of the circle
    */
   public double getCircumference()
   {
      return circumference;
   }
}
```

The important statement in the constructor is

```
super(r);
```

which calls the superclass constructor for the argument `r`. The effect is to construct the superclass part of a `CircleCalculatorB` object. This constructor call must be the first statement in the constructor. If the first statement in the constructor is not a `super` statement the default statement

```
super();
```

is automatically inserted by the compiler. In our case the default statement is not appropriate since our superclass constructor has one argument, the radius of the circle.

In the superclass we could have left the `radius` data field private but then the subclass could not use it directly in the formula for the circumference: the statement

```
circumference = 2.0 * Math.PI * radius;
```

would now be illegal since it tries to access a private data field of another class. However the subclass could still calculate the circumference but the statement

```
circumference = 2.0 * Math.PI * getRadius();
```

would be required.

In the subclass we did not include the `getRadius()` and `getArea()` methods: all public and protected methods of a superclass are automatically available in all subclasses.

The following class can be used to test the two classes.

Class `CircleCalculatorTester`

book-project/chapter9/geometry

```
package chapter9.geometry;
public class CircleCalculatorTester
{
   public void doTest()
```

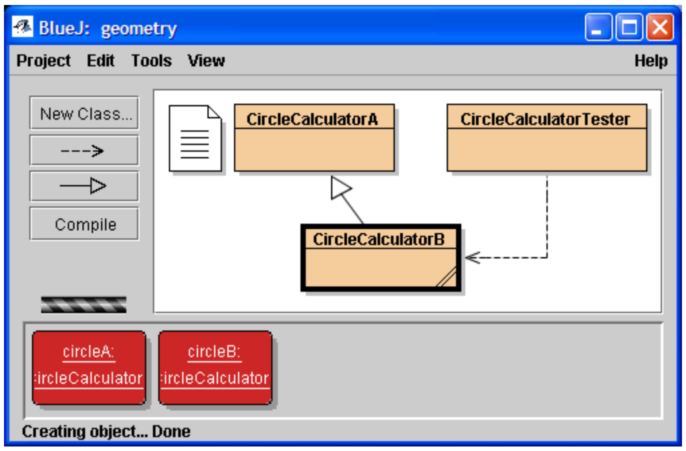


Figure 9.6: Inheritance in BlueJ is indicated by a solid arrow

```
   {
      CircleCalculatorB circle = new CircleCalculatorB(3.0);
      double radius = circle.getRadius();
      double area = circle.getArea();
      double circ = circle.getCircumference();
      System.out.println("Radius:  " + radius);
      System.out.println("Area:  " + area);
      System.out.println("Circumference:  " + circ);
   }

   public static void main(String[] args)
   {
      new CircleCalculatorTester().doTest();
   }
}
```

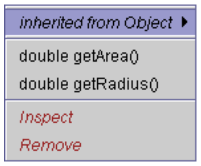
BlueJ project for the circle calculator classes

In BlueJ the inheritance relationship is indicated by a solid arrow from the subclass to the parent class as shown in Figure 9.6 for the classes `CircleCalculatorA` and `CircleCalculatorB`.

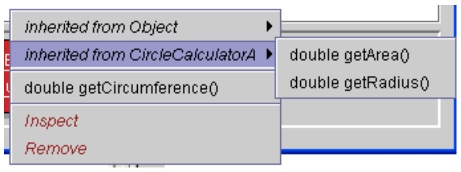
The object menu for a `CircleCalculatorA` object is shown in Figure 9.7(a) and the object menu for a `CircleCalculatorB` object is shown in Figure 9.6(b) as a two level menu. The first level shows the new method that has been added by `CircleCalculatorB` and the second level shows the two methods that have been inherited from `CircleCalculatorA`.

9.4.3 Extending the `BankAccount` class

We now want to make a subclass of the `BankAccount` class from Chapter 6, page 284 which is repeated here:



(a)



(b)

Figure 9.7: (a) `CircleCalculatorA` object menu, (b) `CircleCalculatorB` object menu

Class `BankAccount`

`book-project/chapter9/bank_account`

```
package chapter9.bank_account;
 /**
  * A bank account object encapsulates the account number, owner name, and
  * current balance of a bank account.
  * This version checks for illegal method and constructor arguments.
  */
public class BankAccount
{
   private int number;
   private String name;
   private double balance;

   /**
    * Construct a bank account with given account number,
    * owner name and initial balance.
    * @param accountNumber the account number
    * @param ownerName the account owner name
    * @param initialBalance the initial account balance
    * @throws IllegalArgumentException if account number is negative,
    * owner name is null or empty, or if balance is negative.
    */
   public BankAccount(int accountNumber, String ownerName, double initialBalance)
   {
      if (accountNumber <= 0)
         throw new IllegalArgumentException("Account number must be positive");
      if (ownerName.equals("") || ownerName == null)
         throw new IllegalArgumentException("Owner name not defined");
      if (initialBalance < 0)
         throw new IllegalArgumentException("Balance must be non-negative");
      number = accountNumber;
      name = ownerName;
      balance = initialBalance;
   }

   /**
    * Deposit money in the account.
    * @param amount the deposit amount. If amount <= 0 the
```

```
     * account balance is unchanged.
     * @throws IllegalArgumentException if deposit amount is negative
     */
    public void deposit(double amount)
    {
       if (amount < 0)
          throw new IllegalArgumentException("Invalid amount for deposit");
       balance = balance + amount;
    }

    /**
     * Withdraw money from the account.
     * If account would be overdrawn the account balance is unchanged.
     * @param amount the amount to withdraw.
     * @throws IllegalArgumentException if withdraw amount is invalid
     */
    public void withdraw(double amount)
    {
       if (amount < 0 || amount > balance)
          throw new IllegalArgumentException("Invalid amount for withdraw");
       balance = balance - amount;
    }

    /**
     * Return the account number.
     * @return the account number.
     */
    public int getNumber()
    {
       return number;
    }

    /**
     * Return the owner name.
     * @return the owner name.
     */
    public String getName()
    {
       return name;
    }

    /**
     * Return the account balance.
     * @return the account balance.
     */
    public double getBalance()
    {
       return balance;
    }

    /**
     * string representation of this account.
     * @return string representation of this account.
     */
    public String toString()
    {
        return "BankAccount[" + number + ", " + name + ", " + balance + "]";
    }
 }
```

The new class will be called `JointBankAccount`. It will have a new data field for the name of the joint owner.

Most of the `BankAccount` methods can be used unchanged in the subclass. The only one we need to override is `toString` since the subclass version needs to display the new data field for the joint owner. We need one new method, `getJointName`, to return the name of the joint owner. The class will have the structure

```
public class JointBankAccount extends BankAccount
{
   // new data field for joint owner name goes here
   // constructors go here
   // new getJointName method goes here
   // overridden version of toString goes here.
}
```

For the new data field we can use the declaration

```
private String jointName;
```

The three data fields in the `BankAccount` class are private so they cannot be directly modified by our subclass. However, they can be accessed using the inquiry methods.

Constructors are never inherited so we need to include a constructor that has four arguments, one for each of the four fields. It has the form

```
public JointBankAccount(int accountNumber, String ownerName,
   String jointOwnerName, double initialBalance)
{
   // initialize the four data fields here
}
```

We seem to run into a problem here. We would like to use the assignment statements

```
this.number = accountNumber;
this.name = ownerName;
this.jointName = jointOwnerName;
this.balance = initialBalance;
```

in the body of the constructor but the three fields declared in the superclass are private, since they are managed entirely by the superclass. The way around this is to use `super` to call the superclass constructor and let it initialize the superclass part of a `JointBankAccount` object. This gives the constructor declaration

```
public JointBankAccount(int accountNumber, String ownerName,
   String jointOwnerName, double initialBalance)
{
   super(accountNumber, ownerName, initialBalance); // superclass part
   this.jointName = jointOwnerName; // subclass part
}
```

The `super` statement is a call to the superclass constructor to initialize the superclass part (data fields) of a subclass object. Thus, there is no need for the subclass to directly access or change these data fields.

The overridden `toString` method can also use `super.toString()` to call the superclass version so that it can return its part of the string representation. Then the string expression

```
"JointBankAccount[" + super.toString() + ", " + jointName + "]";
```

can be used to provide a string representation of a subclass object. Here is the complete subclass declaration:

Class `JointBankAccount`

`book-project/chapter9/bank_account`

```
package chapter9.bank_account;
/**
 * A type of BankAccount that includes a joint owner in addition to
 * the owner provided by BankAccount
 */
public class JointBankAccount extends BankAccount
{
   private String jointName;

   /**
    * Construct a joint bank account with given account number,
    * owner name, joint owner name and initial balance.
    * @param accountNumber the account number
    * @param ownerName the account owner name
    * @param jointOwnerName the account joint owner
    * @param initialBalance the initial account balance
    * @throws IllegalArgumentException if account number is negative,
    * owner or joint owner name is null or empty, or if balance is negative.
    */
   public JointBankAccount(int accountNumber, String ownerName, String jointOwnerName,
      double initialBalance)
   {
      super(accountNumber, ownerName, initialBalance);

      if (jointOwnerName.equals("") || jointOwnerName == null)
         throw new IllegalArgumentException("Joint owner name not defined");

      jointName = jointOwnerName;
   }
```

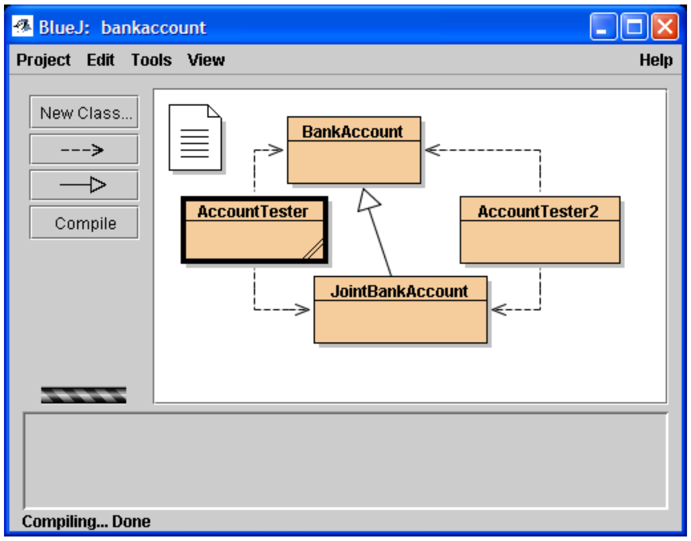


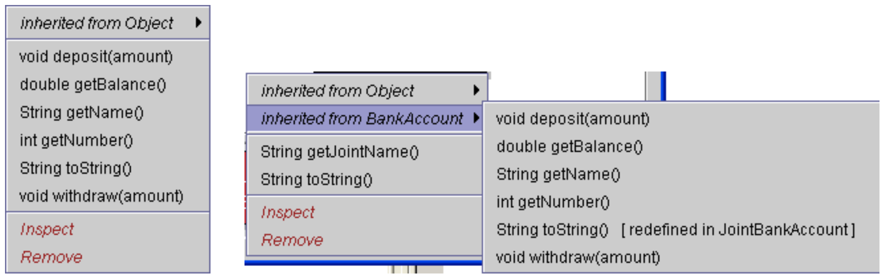
Figure 9.8: Bank account inheritance project

```
    /**
     * Return the joint owner name.
     * @return the joint owner name.
     */
    public String getJointName()
    {
       return jointName;
    }

    /**
     * string representation of this account.
     * @return string representation of this account.
     */
    public String toString()
    {
        return "JointBankAccount[" + super.toString() + ", " + jointName + "]";
    }
}
```

BlueJ project for the bank account classes

To test the two bank account classes they can be placed in a BlueJ project called `bankaccount` as shown in Figure 9.8. The solid arrow indicates that the `JointBankAccount` class is a subclass of the `BankAccount` class.

In Figure 9.9 the object menu for each of the two classes is shown. The `JointBankAccount` object menu at the top level shows only the new `getJointName` method and the overridden `toString` method and the second-level menu shows the methods inherited from the `BankAccount` class and indicates that `toString` was overridden by the `JointBankAccount` class.



(a) (b)

Figure 9.9: (a) `BankAccount` object menu, (b) `JointBankAccount` object menu showing the inherited methods from the `BankAccount` class.

9.5 Polymorphism

We can now introduce polymorphism, one of the most important benefits of inheritance. There are two concepts: polymorphic types and polymorphic methods (the word polymorphism means "many forms").

9.5.1 Polymorphic types

A polymorphic type is a hierarchy of classes defined by inheritance (later we will see that interfaces can also be used to define a polymorphic type). Each class is a subclass of the classes higher up in the hierarchy. Even though the various subclasses are different from each other we can think of them all as being of a similar type, namely the type of their top level superclass. Since `Object` is at the top of any hierarchy this means that every object is a type of `Object`.

This is sometimes called the "is a", "is a kind of", or the "is a type of" relationship. We will see that an important benefit of inheritance is that a superclass object reference can hold a reference to an object of any of its subclasses.

Polymorphic types in the `BankAccount` hierarchy

Consider the following statements:

```
BankAccount fred = new BankAccount(123, "Fred", 345.50);
JointBankAccount fredMary = new JointBankAccount(345, "Fred", "Mary", 450.65);
BankAccount ellenFrank = new JointBankAccount(456, "Ellen", "Frank", 3450.99);
```

The first two statements construct a `BankAccount` object and a `JointBankAccount` object and declare references to them. However, on the right side of the third statement a `JointBankAccount` object is created and its reference is then assigned to `ellenFrank`, which is a `BankAccount` reference (superclass reference). This permits us to treat the `ellenFrank` object as though it were a `BankAccount` object. The converse is not true and the following statement gives a compiler error:

```
JointBankAccount fred = new BankAccount(123, "Fred", 345.50);
```

Thus, we cannot assign a superclass reference to a subclass reference. This is understandable since we cannot say that a bank account object "is a" joint bank account object: it does not contain a joint owner name field.

To use the superclass `getName` method and the subclass `getJointName` method for `fredMary` it is only necessary to write statements such as

```
String owner = fredMary.getName();
String jointOwner = fredMary.getJointName();
```

This follows since `fredMary` is declared as a `JointBankAccount` reference and the `getName` method is inherited from the superclass. However if we do the same with `ellenFrank`, namely

```
String owner = ellenFrank.getName();
String jointOwner = ellenFrank.getJointName();
```

the first statement is fine but the compiler complains for the second statement that there is no `getJointName` method in the `BankAccount` class, which is true.

The `ellenFrank` object is being considered as a `BankAccount` object and has forgotten that it is really a `JointBankAccount` object. This is a form of **object amnesia**: when an object reference is assigned to a superclass reference the object forgets that it really belongs to the subclass. To overcome the object amnesia it is only necessary to write

```
String jointOwner = ((JointBankAccount) ellenFrank).getJointName();
```

which uses a typecast. The extra parentheses are necessary here to indicate that the typecast should be applied to the `ellenFrank` object.

The following simple class illustrates the above ideas and can be used both inside and outside the BlueJ environment.

Class `AccountTester`

`book-project/chapter9/bank_account`

```
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

/**
 * A simple class to illustrate typecasting
 * in the BankAccount hierarchy
 */
public class AccountTester
{
   public void doTest()
   {
      JointBankAccount fredMary = new JointBankAccount(123, "Fred", "Mary", 1000);
      BankAccount ellenFrank = new JointBankAccount(345, "Ellen", "Frank", 1000);

      String jointName1 = fredMary.getJointName();
      String jointName2 = ((JointBankAccount) ellenFrank).getJointName();
      System.out.println("Joint name 1 is " + jointName1);
      System.out.println("Joint name 2 is " + jointName2);
   }

   public static void main(String[] args)
   {
      new AccountTester().doTest();
   }
}
```

Here we construct two `JointBankAccount` objects but the second is assigned to a `BankAccount` reference so it forgets that it is really a `JointBankAccount` object.

Examples of polymorphism

■ EXAMPLE 9.4 **(Object class polymorphism)** The statements

```
Object p = new Point(3,4);
Object c = new Circle((Point) p, 5);
```

show that a `Point` object "is a type of" `Object` and a `Circle` object "is a type of" `Object`. Both objects have forgotten their actual types so the statements

```
System.out.println("x coordinate of p is " + p.getX());
System.out.println("Center of c is " + c.getCenter());
```

give compiler errors since the `Object` class does not have these methods. The statements

```
System.out.println("x coordinate of p is " + ((Point) p).getX());
System.out.println("Center of c is " + ((Circle) c).getCenter());
```

that use a typecast are necessary to remind `p` that it is a `Point` and `c` that it is a `Circle`. ■

■ EXAMPLE 9.5 **(`Graphics` and `Graphics2D`)**
In the `paintComponent` method (see Chapter 5) the statement

```
Graphics2D g2D = (Graphics2D) g;
```

illustrates how a subclass reference can be extracted from a superclass reference. The `Graphics` class is an abstract class for basic graphics methods. Concrete subclasses are available for graphics output devices such as the screen or a printer. When Java 2D was introduced this class was extended to the abstract class `Graphics2D` which, as we have seen in Chapter 5, provides more graphics functionality. When the system calls the `paintComponent` method, whose argument is an object `g` from a subclass of `Graphics`, it actually provides a reference to a subclass instance of `Graphics2D` which implements all the abstract methods. Therefore the `Graphics` reference `g` can be typecast to a `Graphics2D` reference `g2D` which can be used in the `paintComponent` method to access the new Java 2D graphics methods. The original graphics methods can still be accessed using `g` instead of `g2D`. ■

9.5.2 Polymorphic methods

Another form of polymorphism is the ability of an instance method in a class hierarchy to have many different forms, one for each subclass in the hierarchy. Such methods are called **polymorphic methods** and are made possible because we can override superclass methods in subclasses.

Do not confuse method overriding with method overloading, which has nothing to do with polymorphism. Method overloading simply refers to the concept that several methods in the same class can have the same name as long as they have distinguishable argument lists (signatures). In method overriding the methods have the same name and the same argument list but they are in different subclasses.

We will see that objects from the subclasses in an inheritance hierarchy that has polymorphic methods can be processed with these methods in a uniform manner, without regard to the particular subclass of the object, and the particular version of the method.

■ EXAMPLE 9.6 **(`Point2D` superclass)** In graphics programs we used statements such as

```
Point2D.Double bottomRight = new Point2D.Double(300.0, 200.0);
```

Since `Point2D.Double` is a subclass of the `Point2D` class we can shorten this statement to

```
Point2D bottomRight = new Point2D.Double(300.0, 200.0);
```

which just uses `Point2D` on the left side of the assignment. The same applies to the graphics classes such as `Line2D` and `Ellipse2D`. ■

A polymorphic bank account transfer method

Suppose we are processing bank transactions that transfer money from one account to another. We can write a transfer method to do this. Without inheritance we would need four separate methods with the prototypes

```
public void transfer(BankAccount from, BankAccount to, double amount)
public void transfer(BankAccount from, JointBankAccount to, double amount)
public void transfer(JointBankAccount from, BankAccount to, double amount)
public void transfer(JointBankAccount from, JointBankAccount to, double amount)
```

that specify the source account (`from`), the destination account (`to`) and how much to transfer (`amount`), since there are four possibilities for the type of account. Since a `JointBankAccount` is also a `BankAccount` object we need only one method

```
public void transfer(BankAccount from, BankAccount to, double amount)
{
   from.withdraw(amount);
   to.deposit(amount);
}
```

This is made possible by polymorphism. Because the `withdraw` and `deposit` methods are polymorphic within the bank account hierarchy, and because the `transfer` method arguments are declared to be of the base class type, we need only one form of the method.

For example, if `from` is a `JointBankAccount` object then the Java interpreter executes the statement

```
from.withdraw(amount);
```

by first looking for a `withdraw` method in the `JointBankAccount` class. The method is not found so the interpreter goes up the hierarchy one level to the `BankAccount` class, finds the method there, and executes it.

The polymorphic `toString` method

In the bank account hierarchy, each class has its own `toString` method: so this method has three forms, one in the `Object` class, one in the `BankAccount` class, and one in the `JointBankAccount` class. Therefore `toString` is a **polymorphic method**. The following simple class illustrates this idea.

Class `AccountTester2`

`book-project/chapter9/bank_account`

```
package chapter9.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

/**
 * A simple class to illustrate the polymorphic toString
 * method in the BankAccount hierarchy
 */
public class AccountTester2
{
   public void doTest()
   {
      BankAccount fred = new BankAccount(456, "Fred", 500);
      JointBankAccount fredMary = new JointBankAccount(123, "Fred", "Mary", 1000);
      BankAccount ellenFrank = new JointBankAccount(345, "Ellen", "Frank", 1000);

      System.out.println(fred);
      System.out.println(fredMary);
      System.out.println(ellenFrank);
   }

   public static void main(String[] args)
   {
      new AccountTester2().doTest();
   }
}
```

The output is

```
BankAccount[456, Fred, 500.0]
JointBankAccount[BankAccount[123, Fred, 1000.0], Mary]
JointBankAccount[BankAccount[345, Ellen, 1000.0], Frank]
```

Even though the third account is assigned to the superclass `BankAccount` reference the run-time system knows that the account is really a `JointBankAccount` so the `println` method calls the `toString` method in this class.

Compile-time and run-time types

It is important to understand the difference between the **compile-time type** and the **run-time type** of an object. The compile-time type is the type given explicitly in the class. This may or may not also be the run-time type which is the actual type obtained using `new` and known to the run-time system (Java Virtual Machine).

For example in the above `AccountTester2` class `fred` has `BankAccount` as both its compile-time and run-time type. Similarly, `fredMary` has `JointBankAccount` as its compile-time and run-time type. However, `ellenFrank` has compile-time type `BankAccount` but run-time type `JointBankAccount`. When `println` is executed at run-time it is always the run-time type that is used.

9.6 Abstract classes and polymorphism

We now introduce the concept of an **abstract class** and illustrate it with a simple employee inheritance hierarchy. An abstract class is a class that declares at least one method without providing a method body, i.e., no implementation is defined, only the method prototype. Each such method is called an **abstract method**. The class is specified using the `abstract` keyword.

When you specify an abstract method you are forcing each non-abstract subclass to provide an implementation (method body) for it having exactly the specified prototype. Thus, each abstract method is polymorphic.

9.6.1 An employee inheritance hierarchy

We now develop Java classes for the employee hierarchy given in Example 9.3 and Figure 9.3. The five classes can be described as follows:

Employee An abstract class that encapsulates the name of the employee. It has two abstract methods: `grossSalary` calculates and returns the gross monthly salary, and `netSalary` calculates and returns the net monthly salary (salary after deductions). It also has a `getName` method to return the name and a `toString` method. It will have a constructor with the prototype

```
public Employee(String name)
```

Manager An employee with a gross monthly salary from which 10% is deducted to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```
public Manager(String name, double salary)
```

HourlyWorker An employee whose gross monthly salary is determined by the number of hours worked and the hourly wage. From this gross amount 5% is deducted to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```
public HourlyWorker(String name, double hoursWorked, double hourlyRate)
```

PartTimeWorker An employee like an hourly worker but with no deductions to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```
public PartTimeWorker(String name, double hoursWorked, double hourlyRate)
```

CommissionWorker An employee who receives a base monthly salary like a manager but a sales bonus is added to get the gross monthly salary. The bonus is a specified percentage of monthly sales. Thus, the gross monthly salary is

```
(base salary) + (monthly sales) * (commission rate in percent / 100.0).
```

From this 10% is deducted to get the net monthly salary. It will implement the `grossSalary` and `netSalary` methods, provide a `toString` method, and have a constructor with the prototype

```
public CommissionWorker(String name, double baseSalary,
   double monthlySales, double commissionRate)
```

9.6.2 `Employee` and `Manager` classes

We give declarations for the first two classes here and the remaining three classes are left as an exercise (see Exercise 9.3).

Class `Employee`

book-project/chapter9/employee

```
package chapter9.employee;
/**
 * An abstract class representing an employee.
 * The abstract grossSalary and netSalary methods are polymorphic.
 */
abstract public class Employee
{
   private String name;

   /** Construct the name part of an employee.
    * @param name the name part of an employee
    */
   public Employee(String name)
   {
      this.name = name;
   }

   /** Return the employee name.
    * @return the employee name
    */
   public String getName()
   {
      return name;
   }

   /** Return the gross salary of an employee.
    * @return gross salary of an employee
    */
   abstract public double grossSalary();

   /** Return the net salary of an employee.
    * @return net salary of an employee
    */
   abstract public double netSalary();
}
```

Since this is an abstract class we cannot construct an `Employee` object so you may wonder why we have declared a constructor in the class. The reason is so that subclasses can use `super` to call the superclass `Employee` constructor to initialize the private data field for the employee name (see `Manager` class below).

Non-abstract subclasses of `Employee` must implement the two abstract methods. For `Manager` we have the class declaration

Class `Manager`

`book-project/chapter9/employee`

```
package chapter9.employee;
/**
 * A class for employees that are managers.
 * The abstract grossSalary and netSalary methods are implemented
 * and the toString method is overridden.
 */
public class Manager extends Employee
{
   private double gross; // gross monthly salary
   private double net;   // net monthly salary

   /** Construct a manager object with given name and salary
    * @param name the name of the manager
    * @param salary the gross salary of a manager
    */
   public Manager(String name, double salary)
   {
      super(name); // superclass is responsible for name
      gross = salary;
      net = 0.9 * gross;
   }

   /** Return the gross salary of a manager.
    * @return the gross salary of a manager
    */
   public double grossSalary()
   {
      return gross;
   }

   /** Return the net salary of a manager.
    * @return the net salary of a manager
    */
   public double netSalary()
   {
      return net;
   }

   /** Return the string representation of a manager.
    * @return the string representation of a manager
    */
   public String toString()
   {
      return "Manager[" + "name = " + getName() +
         ", gross = " + grossSalary() + ", net = " + netSalary() + "]";
   }
}
```

9.6.3 Polymorphism in the `Employee` hierarchy

There are three polymorphic methods in the `Employee` hierarchy: `grossSalary`, `netSalary`, and `toString`. Each of the four subclasses of `Employee` has its own version of these methods. We can illustrate polymorphism by writing a class that stores some objects in an array of `Employee` references and uses a loop to compute the total gross monthly salary of all employees, the total net monthly salary of all employees, and the total deductions. Here is a tester class containing a `main` method.

Class `EmployeeProcessor`

`book-project/chapter9/employee`

```
package chapter9.employee;
/**
 * Illustrate polymorphism in the Employee hierarchy.
 */
public class EmployeeProcessor
{
   private Employee[] staff;
   private double totalGrossSalary;
   private double totalBenefits;
   private double totalNetSalary;

   /** Process an array of 5 employees and compute totals
    * for gross salary, net salary, and benefits.
    */
   public void doTest()
   {
      staff = new Employee[5];
      staff[0] = new Manager("Fred", 800);
      staff[1] = new Manager("Ellen", 700);
      staff[2] = new HourlyWorker("John", 37, 13.50);
      staff[3] = new PartTimeWorker("Gord", 35, 12.75);
      staff[4] = new CommissionWorker("Mary", 400, 15000, 3.5);

      /* Compute the total gross salary, net salary and benefits for all
         employees without knowing the kinds of employees when we write
         the class.
      */
      totalGrossSalary = 0.0;
      totalNetSalary = 0.0;
      for (int i = 0; i < staff.length; i++)
      {
         totalGrossSalary = totalGrossSalary + staff[i].grossSalary();
         totalNetSalary = totalNetSalary + staff[i].netSalary();
         System.out.println(staff[i]);
      }
      totalBenefits = totalGrossSalary - totalNetSalary;
      System.out.println("Total gross salary: " + totalGrossSalary);
      System.out.println("Total benefits: " + totalBenefits);
      System.out.println("Total net salary: " + totalNetSalary);
   }

   public static void main(String[] args)
   {
      new EmployeeProcessor().doTest();
   }
}
```

The constructor first creates an array called `staff` of 5 references to base class `Employee` objects. They can refer to objects of any subclass so the next step is to construct five subclass objects and assign their references to the array elements. Finally, a simple loop, using the polymorphic `grossSalary`, `netSalary` and `toString` methods, displays the employee information and computes the total gross and net salaries of all employees. This is possible because these two methods were declared abstract in the `Employee` class so all the non-abstract subclasses are guaranteed to have implementations of them. Here is the output.

```
Manager[name = Fred, gross = 800.0, net = 720.0]
Manager[name = Ellen, gross = 700.0, net = 630.0]
HourlyWorker[name = John, gross = 499.5, net = 474.525]
PartTimeWorker[name = Gord, gross = 446.25, net = 446.25]
CommissionWorker[name = Mary, gross = 925.0, net = 832.5]
```

```
Total gross salary: 3370.75
Total benefits: 267.4749999999999
Total net salary: 3103.275
```

There are two important ideas here. The first is that it is not necessary to know anything about the kind of employee being processed with each loop iteration when the class is written. The system determines at run-time which kind of object is being used so the appropriate version of each polymorphic method is selected. The second is that if new kinds of employees are added to the hierarchy it is not necessary to make any modifications to the polymorphic loop that calculates the total gross and net salaries.

9.7 The `Object` class

The `Object` class is the ultimate parent of any Java class: an object of any Java class “is a type of” `Object`. This class declares several useful methods that are automatically inherited by any class. Here is a partial specification of the `Object` class

```
public class Object
{
   public Object() {...}
   public String toString() {...}
   public boolean equals(Object obj) {...}
   protected Object clone() {...}
   Class <? extends Object> getClass() {...}
   int hashCode() {...}
   // several other methods
}
```

We have already used the `toString` method and will discuss the others as needed.

9.7.1 Overriding `Object` class methods

Since the `Object` class is a superclass of all classes, any public or protected methods that it contains are automatically available to any class or can be overridden.

Overriding the `toString` method

For example, if we do not include a `toString` method in our classes we can still use it, since ultimately the `Object` class version will be called. This explains the output in Examples 4.15, 4.16, and 4.17 from Chapter 4. Without a `toString` method the `doTest` method in `EmployeeProcessor` would produce the output

```
Manager@310d42
Manager@5d87b2
HourlyWorker@77d134
PartTimeWorker@47e553
```

```
CommissionWorker@20c10f
Total gross salary: 3370.75
Total benefits: 267.4749999999999
Total net salary: 3103.275
```

The strings produced are not very useful however; just the name of the class and a hexadecimal number. This is understandable since the `Object` class method does not know much about the `Employee` and `Manager` classes and other classes in the hierarchy. Therefore most classes override `toString` to provide a more meaningful string representation of an object.

Overriding the `equals` method

Similarly, the `equals` method provided in the `Object` class is not very useful since it just compares two references rather than the objects referenced. The `Object` class has no idea what kind of objects you are using and what your definition of equality is, so subclasses normally override it. For example, the `String` class in package `java.lang` has its own version of `equals` which we have used many times to compare two strings lexicographically.

To illustrate the `equals` method we can use the `Point` class from Chapter 4, page 119. Let us add an `equals` method to this class that compares two points using the definition that two `Point` objects are equal if both their *x* and *y*-coordinates are equal.

To test the `equals` method the new `Point` class has the structure

```
public class Point
{
   double x, y;
   public Point(double x, double y) { this.x = x; this.y = y; }
   public Point() { x = 0.0; y = 0.0; }
   public double getX() { return x; }
   public double getY() { return y; }
   public String toString() { return "Point[" + x + ", " + y + "]"; }

   public boolean equals(Object obj) { ... }
}
```

This class and the following `PointEqualsTester` class are in a package called `chapter9.equals`.

Class `PointEqualsTester`

book-project/chapter9/equals

```
package chapter9.equals;
/**
 * A tester class for equals method in Point class
 */
public class PointEqualsTester
{
   public void doTest()
   {
      Point p = new Point(3,4);
      Point q = new Point(3,4);
      Point r = new Point(3,5);
      if (p.equals(q))
         System.out.println("p and q are equal");
      else
         System.out.println("p and q are not equal");

      if (q.equals(r))
         System.out.println("q and r are equal");
      else
         System.out.println("q and r are not equal");
   }

   public static void main(String[] args)
   {
      new PointEqualsTester().test();
   }
}
```

First compile the `Point` and `PointTester` classes without an `equals` method. This forces the `Object` version to be used. The output from the tester class is

```
p and q are not equal
q and r are not equal
```

which simply tells us that the three references `p`, `q`, and `r` are different. This is not very useful. We would like to have `p.equals(q)` return true to indicate that, even though the references `p` and `q` are different, the two objects are equal. Here are two versions of the `equals` method.

The first is

```
public boolean equals(Point p)
{
  if (obj instanceof Point)
  {
     Point p = (Point) obj;
     return (x == p.x && y == p.y);
  }
  return super.equals(obj);
}
```

which uses the `instanceof` operator to check if `obj` has the correct type. Otherwise the type cast would throw a `ClassCastException`.

The second is

```
public boolean equals(Object obj)
{
   if (obj == null) return false;
   if (! this.getClass().equals(obj.getClass())) return false;
   Point p = (Point) obj;
   return (x == p.x && y == p.y);
}
```

where we use the `getClass` method in the `Object` class to test if two objects have the same class. In our case the if statement compares the class of `this` object with the class of `obj`. If they are not the same then `false` is returned. Then the statement with the typecast will be executed only if `obj` really is a `Point` object. A value of `true` will be returned only if the `x` and `y` coordinates of the two points are the same.

With either version of `equals` the output of the tester class is

```
p and q are equal
q and r are not equal
```

which shows that the point objects are being compared, not their references.

The version using `instanceof` is the correct one if the class is declared to be final (see below). Otherwise the version using `getClass` should be used.

9.8 Final classes

A final class is one that cannot be extended so it can have no subclasses. The **`final`** keyword is used to indicate that a class is final. For example

```
public final MyFinalClass
{
   // ...
}
```

is a final class so it would be a compiler error to try to write a class such as

```
public final MySubClass extends MyFinalClass
{
   // ...
}
```

Final classes are usually more efficient. Also you gain more control over a class if its final since no one can override the methods in a final class. Many of the standard classes are final for efficiency and security. For example the `String` class is used everywhere and is declared to be final so it is impossible to override the `length` method and provide an incorrect value.

9.9 Interfaces

An **interface** is a kind of purely abstract class. It can contain only method prototypes and constants. No implementation of any of the methods can be provided. It is declared like a class using the keyword **`interface`** instead of the keyword `class`:

```
public interface MyInterface
{
   // method prototypes go here, if any
}
```

Unlike abstract classes, interfaces cannot declare constructors since there are no objects to construct. Since all methods in an interface are abstract the `abstract` keyword is not needed. Also, every method in an interface must be public so the `public` keyword is also redundant but often included.

To make use of an interface we need to provide classes that "implement the interface". Such classes must provide complete declarations (implementations) for all the methods declared by the interface. If `MyClass` is a class that implements an interface called `MyInterface` then the syntax of this class declaration is

```
public class MyClass implements MyInterface
{
   // data fields
   // constructors
   // methods not related to interface, if any
   // Implementations of the interface methods
}
```

Here the keyword **`implements`** is used instead of **`extends`**.

The important idea here is that we can say that an object of `MyClass` "is of type" `MyInterface`. This is possible since we can declare interface references and assign to them references to any object of any class that implements the interface. For example suppose `MyClass1` and `MyClass2` are classes that implement `MyInterface`. Then with the declarations

```
MyInterface myObject1 = new MyClass1(...);
MyInterface myObject2 = new MyClass2(...);
```

`myObject1` "is a type of" `MyInterface` and so is `myObject2`. This means that polymorphism also applies to interfaces in the sense that the classes that implement an interface form a polymorphic type and the interface methods are polymorphic.

However, interfaces can be more flexible and general than class inheritance hierarchies since the classes that implement an interface do not need to be related in any other way. In particular, they do not have to belong to any class inheritance hierarchy.

It is also possible to have interface inheritance hierarchies. For example, we can extend `MyInterface` to obtain a subinterface called `MySubinterface` using

```
public interface MySubinterface extends MyInterface
{
   // optionally the MyInterface prototypes can be included here
   // new method prototypes are included here
}
```

If a class wants to implement `MySubinterface` it must implement all the methods in `MyInterface` as well as the new ones in `MySubinterface`. The method prototypes in `MyInterface` can also be repeated in `MyInterface`.

Multiple inheritance is allowed for interfaces in the sense that a class can implement several interfaces but extend only one class. Therefore the general structure of a class declaration that extends another class and implements several interfaces is

```
public class MyClass extends MySuperclass
   implements MyInterface1, MyInterface2, ..., MyInterfaceN
{
   // MyClass data fields
   // MyClass constructors
   // MyClass methods not related to the interfaces, if any
   // Implementations of all interface methods
}
```

which indicates that `MyClass` extends `MySuperclass` and implements *N* interfaces. Two interfaces can have a method with the same name and argument types. Of course any class that implements both interfaces can only provide one implementation of the common method. If this implementation does not make sense for both interfaces then it is not possible to implement both interfaces with one class.

■ EXAMPLE 9.7 **(`Measurable` interface)** The interface

```
public interface Measurable
{
   public double area();
   public double perimeter();
}
```

declares two abstract methods called `area` and `perimeter` that are supposed to represent the area and perimeter of a two-dimensional geometric object. Notice that the method prototypes are terminated by a semi-colon to indicate that there is no implementation. This example illustrates that an interface is a design specification. It specifies what it means for an object to be "measurable": it means that the area and perimeter of the object can be calculated. ■

■ EXAMPLE 9.8 **(`Scalable` interface)** The interface

```
public interface Scalable
{
   public void scale(double s);
}
```

declares one abstract method called `scale` that is supposed to scale a two-dimensional geometric object by the factor `s` in both directions. An object of any class that implements this interface is said to be a `Scalable` object. ■

■ EXAMPLE 9.9 **(Extending the `Scalable` interface)** The interface

```
public interface Scalable2D extends Scalable
{
   public void scale(double sx, double sy);
}
```

extends the Scalable interface by providing the prototype for a more general scale method that can scale using different factors in each direction. This sets up an interface inheritance hierarchy. A class that implements the Scalable2D interface must implement both versions of the scale method. Objects of this class are Scalable2D objects and through inheritance they are also Scalable objects. ■

9.9.1 Implementing the Measurable interface

Let us illustrate interface polymorphism by writing geometric Circle and Rectangle classes that implement the Measurable interface in Example 9.7. A common error is to omit the 'implements' clause on the class declaration. The class will still compile but its objects will not be measurable.

A measurable circle class

The following simple Circle class (see Chapter 4, page 122 for a related class that uses a Point object for the circle center) implements the Measurable interface by providing implementations of the area and perimeter interfaces.

Class Circle

book-project/chapter9/interfaces

```
package chapter9.interfaces;
/**
 * A class for measurable circles
 */
public class Circle implements Measurable
{
   private double x, y;   // coordinates of center
   private double radius;

   public Circle()
   {
      this(0,0,1);
   }

   public Circle(double xc, double yc, double r)
   {
      x = xc;
      y = yc;
      radius = r;
   }

   public double getX()
   {
      return x;
   }

   public double getY()
   {
      return y;
   }

   public double getRadius()
   {
      return radius;
   }

   public String toString()
   {
      return "Circle[x = " + x + ", y = " + y + ", radius = " + radius + "]";
   }

   // Implement the Measurable interface

   public double area()
   {
      return Math.PI * radius * radius;
   }

   public double perimeter()
   {
      return 2.0 * Math.PI * radius;
   }
}
```

If `c` is a `Circle` object we can say that `c` "is of type" `Measurable`, or `c` is a `Measurable` object. In this case we could define a circle `c1` using

```
Circle c1 = new Circle(0.0, 0.0, 1.0);
```

or we could define a circle `c2` using

```
Measurable c2 = new Circle(0.0, 0.0, 1.0);
```

We use this statement in situations where we are only interested in the interface methods: with `c1` we have access to all the methods of the `Circle` class, including those in the interface, but with `c2` we have access, without a typecast, to only the `area` and `perimeter` methods. For example, the last of the statements

```
double a1 = c1.area();
double r1 = c1.getRadius();
double a2 = c2.area();
double r2 = c2.getRadius();
```

gives an error since `c2`, as a `Measurable` object, has forgotten that it is also a `Circle` object (object amnesia again). To fix this we need to typecast:

```
double r2 = ((Circle)c2).getRadius();
```

to remind `c2` that it is also a `Circle` object.

A measurable rectangle class

The following simple `Rectangle` class implements the `Measurable` interface since it provides implementations of the `area` and `perimeter` interfaces.

Class `Rectangle`

`book-project/chapter9/interfaces`

```
package chapter9.interfaces;
/**
 * A class for measurable rectangles
 */
public class Rectangle implements Measurable
{
   private double x, y;            // coordinates of lower left corner
   private double width, height;   // width and height of rectangle

   public Rectangle()
   {
      this(0,0,1,1);
   }

   public Rectangle(double x, double y, double w, double h)
   {
      this.x = x;
      this.y = y;
      width = w;
      height = h;
   }

   public double getX()
   {
      return x;
   }

   public double getY()
   {
      return y;
   }

   public double getWidth()
   {
      return width;
   }

   public double getHeight()
   {
      return height;
   }

   public String toString()
   {
      return "Rectangle[x = " + x + ", y = " + y +
         ", width = " + width + ", height = " + height + "]";
   }

   // Implement the Measurable interface here

   public double area()
   {
      return width * height;
   }

   public double perimeter()
   {
      return 2.0 * (width + height);
   }
}
```

Polymorphism with the `Measurable` interface

The following class shows how to use a polymorphic loop to compute the total area and perimeter of some measurable objects.

Class `MeasurableTester`

`book-project/chapter9/interfaces`

```
package chapter9.interfaces;
/**
 * Illustrating polymorphism with the Measurable interface
 */
public class MeasurableTester
{
   private Measurable[] a = new Measurable[3];

   public void test()
   {
      a[0] = new Circle(0,0,1);
      a[1] = new Circle(1,1,2);
      a[2] = new Rectangle(5,5,20,10);

      double areaSum = 0.0;
      double perimeterSum = 0.0;
      for (int k = 0; k < a.length; k++)
      {
         areaSum = areaSum + a[k].area();
         perimeterSum = perimeterSum + a[k].perimeter();
         System.out.println(a[k]);
         System.out.println("Perimeter = " + a[k].perimeter()
           + ", Area = " + a[k].area());
      }
      System.out.println("Total area is " + areaSum);
      System.out.println("Total perimeter is " + perimeterSum);
   }

   public static void main(String[] args)
   {
      new MeasurableTester().test();
   }
}
```

The program declares an array of type `Measurable` and then assigns references to `Circle` and `Rectangle` objects to the array elements. The area and perimeter of these objects can then be calculated in a polymorphic loop since any `Measurable` object has `perimeter` and `area` methods. It is important that `a` is an array of `Measurable` type. The program output is

```
Circle[x = 0.0, y = 0.0, radius = 1.0]
Perimeter = 6.283185307179586, Area = 3.141592653589793
Circle[x = 1.0, y = 1.0, radius = 2.0]
Perimeter = 12.566370614359172, Area = 12.566370614359172
Rectangle[x = 5.0, y = 5.0, width = 20.0, height = 10.0]
Perimeter = 60.0, Area = 200.0
Total area is 215.70796326794897
Total perimeter is 78.84955592153875
```

9.9.2 Polymorphism with the `Shape` interface

The graphics hierarchy shown in Chapter 5, Figure 5.6, is a hierarchy in which classes such as `Line2D` and `Rectangle2D` implement the `Shape` interface. Therefore we can say that a `Line2D` object “is of type” `Shape`. The `Shape` interface simply specifies the methods that a class must implement in order to be called a `Shape`. Also, the `RectangularShape` class is an example of an abstract class that implements the `Shape` interface. Its purpose is to provide a base class for the classes that are specified using bounding rectangular boxes in their description. These are the `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, and `Arc2D` classes. The polymorphic `draw` and `fill` methods in the `Graphics2D` class have prototypes

```
public void draw(Shape s);
public void fill(Shape s);
```

so they can take as an argument an object of any class, such as `Rectangle2D`, that implements the `Shape` interface. Without the `Shape` polymorphism we would need separate draw and fill commands for each kind of shape: e.g., `drawLine`, `drawRect`, `drawEllipse`, and so on.

We can illustrate polymorphism in the set of classes implementing the `Shape` interface. `Shape` is an interface in package `java.awt` that declares 10 rather complicated methods that are needed to draw and fill shapes. Classes such as `Line2D.Double` and `Rectangle2D.Double` implement this interface so each object from one of these classes is a type of `Shape`. Previously we defined graphics objects using declarations such as

```
Line2D.Double line = new Line2D.Double(0.0,0.0,200.0,150.0);
Rectangle2D.Double rect = new Rectangle2D.Double(10.0,10.0,100.0,150.0);
```

Since `line` and `rect` are each a type of `Shape`, instead of these declarations we could have used

```
Shape line = new Line2D.Double(0.0,0.0,200.0,150.0);
Shape rect = new Rectangle2D.Double(10.0,10.0,100.0,150.0);
```

Doing it this way permits a polymorphic processing of graphical objects using the `draw` and `fill` methods.

If we have an array of type `Shape` and assign various objects to it from classes that implement the `Shape` interface then we can write a single polymorphic loop to process all the objects using `draw` and `fill`. For example, let us declare an array of `Shape` references:

```
Shape[] shape = new Shape[5];
```

This is legal: even though there is no such thing as a `Shape` object we can declare an array of references to objects from classes that implement the `Shape` interface. Therefore, we can use statements such as

```
shape[0] = new Line2D.Double(0.0,0.0,200.0,150.0);
```

since a `Line2D.Double` object is a type of `Shape`. We can now draw all the shapes in one polymorphic loop such as

```
for (int k = 0; k < shape.length; k++)
{
   g2D.draw(shape[k]);
}
```

without knowing the particular kinds of shape. Without polymorphism we would have to use a giant if statement inside the loop (if the object is of type T1 draw it this way, else if it is of type T2 draw it that way, else, ...). Here is a simple graphics program to illustrate this uniform processing of geometrical objects.

Class `ShapeTester`

`book-project/chapter9/shapetest`

```
package chapter9.shapetest;
import custom_classes.GraphicsFrame;
/**
 * Illustrating polymorphism within the Shape hierarchy.
 */
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class ShapeTester extends JPanel
{
   Shape[] shape = new Shape[5]; // array to hold Shape objects

   public ShapeTester()
   {
      shape[0] = new Line2D.Double(0.0,0.0,200.0,150.0);
      shape[1] = new Rectangle2D.Double(10.0,10.0,100.0,50.0);
      shape[2] = new RoundRectangle2D.Double(120.0,20.0,60.0,30.0,40.0,40.0);
      shape[3] = new Ellipse2D.Double(30.0,70.0,50.0,50.0);
      shape[4] = new Ellipse2D.Double(130.0,100.0,50.0,25.0);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      // following statements resize the graphics when the frame is resized

      double xMax = getWidth() - 1;
      double yMax = getHeight() - 1;
      AffineTransform at = new AffineTransform();
      at.translate(xMax / 2, yMax / 2);
      at.scale(xMax / 200, yMax / 150);
      at.translate(-100,-75);
      g2D.transform(at);

      for (int k = 0; k < shape.length; k++)
      {
         g2D.setPaint(Color.pink);
         g2D.fill(shape[k]);
         g2D.setPaint(Color.black);
         g2D.draw(shape[k]);
      }
   }

   public void draw()
   {
      new GraphicsFrame("Some shapes", this, 201, 151);
   }

   public static void main(String[] args)
   {
      new ShapeTester().draw();
   }
}
```

The output is shown in Figure 9.10.

9.10 Multiple interfaces

It is possible for a class to implement more than one interface. As an example we consider three interfaces that define methods useful in working with two-dimensional geometrical objects.

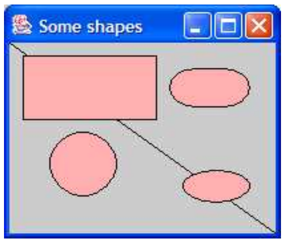


Figure 9.10: Output of `ShapeTester` class

9.10.1 Interface specifications

The first is the `Measurable` interface. It declares prototypes for methods that calculate the area and perimeter of a geometrical object. The interface declaration is

Interface `Measurable`

`book-project/chapter9/multiple_interfaces`

```
package chapter9.multiple_interfaces;
/**
 * An interface for geometric objects that have
 * an area and perimeter
 */
public interface Measurable
{
   public double area();
   public double perimeter();
}
```

Similarly, we declare a `Translatable` interface which declares the prototype for a method that translates an object by a given distance in the *x* and *y* directions. The interface declaration is

Interface `Translatable`

`book-project/chapter9/multiple_interfaces`

```
package chapter9.multiple_interfaces;
/**
 * An interface for geometric objects that can
 * be translated in a given direction
 */
public interface Translatable
{
   public void translate(double dx, double dy);
}
```

Finally we define a `Scalable` interface which declares the prototype for a method that scales an object by a given amount (zoom factor) in both directions. The interface declaration is

Interface `Scalable`

book-project/chapter9/multiple_interfaces

```
package chapter9.multiple_interfaces;
/**
 * An interface for geometric objects that can
 * be scaled (made larger or smaller)
 */
public interface Scalable
{
   public void scale(double s);
}
```

9.10.2 Classes that implement the interfaces

We can now design classes of geometrical objects that implement one or more of these interfaces. As an example we modify our `Circle` and `Rectangle` classes from Section 9.9.1 so that they implement all three interfaces as follows

Class `Circle`

book-project/chapter9/multiple_interfaces

```
package chapter9.multiple_interfaces;
/**
 * A class that implements multiple interfaces
 */
public class Circle implements Measurable, Translatable, Scalable
{
   private double x, y;   // coordinates of center
   private double radius;

   public Circle(double xc, double yc, double r)
   {
      x = xc;
      y = yc;
      radius = r;
   }

   // implementations of getX, getY, getRadius, toString go here

   // Implement the three interfaces

   public double area()
   {
      return Math.PI * radius * radius;
   }

   public double perimeter()
   {
      return 2.0 * Math.PI * radius;
   }

   public void translate(double dx, double dy)
   {
      x = x + dx;
      y = y + dy;
   }

   public void scale(double s)
   {
      radius = radius * s;
   }
}
```

Class `Rectangle`

book-project/chapter9/multiple_interfaces

```
package chapter9.multiple_interfaces;
/**
 * A class that implements multiple interfaces
 */
public class Rectangle implements Measurable, Translatable, Scalable
{
   private double x, y;            // coordinates of lower left corner
   private double width, height;   // width and height of rectangle

   public Rectangle(double x, double y, double w, double h)
   {
      this.x = x;
      this.y = y;
      width = w;
      height = h;
   }

   // implementations of getX, getY, getWidth, getHeight, toString go here

   // Implement the interfaces

   public double area()
   {
      return width * height;
   }

   public double perimeter()
   {
      return 2.0 * (width + height);
   }

   public void translate(double dx, double dy)
   {
      x = x + dx;
      y = y + dy;
   }

   public void scale(double s)
   {
      width = width * s;
      height = height * s;
   }
}
```

9.10.3 Typecasts with multiple interfaces

Suppose we want to do multiple polymorphism: process objects using methods of more than one interface. We can use the `Measurable`, `Translatable`, and `Scalable` interfaces as an example. The program `MeasurableTester` processed `Circle` and `Rectangle` objects using only the `Measurable` interface, so we declared an array

```
      private Measurable[] a = new Measurable[3];
```

and used a single loop to calculate the area and perimeter of the objects. Similarly, if we just wanted to scale the objects we could declare an array

```
      private Scalable[] a = new Scalable[3];
```

Suppose that we want to both scale and translate objects in a single loop. We cannot declare a `Scalable` array since it would be an error to apply `translate` to a `Scalable` object and conversely we cannot declare a `Translatable` array since it would be an error to apply `scale` to a `Translatable` object.

Another solution is to use an array of type `Object`:

```
      private Object[] a = new Object[3];
```

and assign `Circle` and `Rectangle` object references to it. Then we can write one loop to process these objects. The only problem is that the `Circle` and `Rectangle` objects suffer amnesia when they are assigned to the `Object` array. Therefore, when we want to apply `translate` to an object we must remind the object, using a typecast, that it is `Translatable`. Similarly, when we want to apply `scale` we must use a typecast to `Scalable`. Here is a short program that illustrates this multiple polymorphism:

Class `MultipleInterfaceTester`

`book-project/chapter9/multiple_interfaces`

```
package chapter9.multiple_interfaces;
/**
 * Illustrate multiple polymorphism of interfaces
 */
public class MultipleInterfaceTester
{
   private Object[] a = new Object[3];
   public void test()
   {
      a[0] = new Circle(0,0,1);
      a[1] = new Circle(1,1,2);
      a[2] = new Rectangle(5,5,20,10);

      for (int k = 0; k < a.length; k++)
      {
         ((Translatable) a[k]).translate(1,1);
         ((Scalable) a[k]).scale(2);
         System.out.println(a[k]);
      }
   }

   public static void main(String[] args)
   {
      new MultipleInterfaceTester().test();
   }
}
```

In the first statement in the loop `a[k]` is typecast using

```
(Translatable) a[k]
```

to make a `Translatable` object. Now we can apply the `translate` method to this object using

```
((Translatable) a[k]).translate(1,1);
```

The extra set of parentheses are necessary since "dot" has a higher precedence than the typecast. Similarly, `a[k]` is typecast to `Scalable` so that the `scale` method can be applied. The output is

```
Circle[x = 1.0, y = 1.0, radius = 2.0]
Circle[x = 2.0, y = 2.0, radius = 4.0]
Rectangle[x = 6.0, y = 6.0, width = 40.0, height = 20.0]
```

9.11 Implementing the Shape interface

In Chapter 5 we used several classes such as `Rectangle2D` and `Ellipse2D` that we could draw and fill using the polymorphic `draw` and `fill` methods that accept `Shape` arguments. We now show how to create our own classes that implement the `Shape` interface.

9.11.1 Shape interface methods

If you look at the Java documentation you will see that the `Shape` interface declares the following 10 methods:

```
public boolean contains(Point2D p);
public boolean contains(Rectangle2D r);
```

```
      public boolean contains(double x, double y);
      public boolean contains(double x, double y, double w, double h);
      public Rectangle getBounds();
      public Rectangle2D getBounds2D();
      public PathIterator getPathIterator(AffineTransform at);
      public PathIterator getPathIterator(AffineTransform at, double flatness);
      public boolean intersects(Rectangle2D r);
      public boolean intersects(double x, double y, double w, double h);
```

You probably have no idea what many of these methods mean, let alone how to implement them. The good news is that you don't have to know. Since the `GeneralPath` class implements them we can design our own custom graphics classes that can be used as arguments to `draw` and `fill`. There are two ways to do this: using an adapter class that implements `Shape` or implementing the `Shape` interface directly.

9.11.2 Extending a `ShapeAdapter` class which implements `Shape`

Consider the following adapter class.

Class `ShapeAdapter`

book-project/chapter9/shapes

```
package chapter9.shapes;
import java.awt.*;
import java.awt.geom.*;

/**
 * Extend this class to implement the Shape interface for
 * defining geometrical objects.
 */
public class ShapeAdapter implements Shape
{
   /** The path used to define the Shape */
   protected GeneralPath path;

   /** Construct an empty path */
   public ShapeAdapter()
   {
      path = new GeneralPath();
   }

   /*
      implementation of the Shape interface using the
      implementation provided by GeneralPath.
   */

   public boolean contains(Point2D p)
   { return path.contains(p); }
   public boolean contains(Rectangle2D r)
   { return path.contains(r); }
   public boolean contains(double x, double y)
   { return path.contains(x,y); }
   public boolean contains(double x, double y, double w, double h)
   { return path.contains(x,y,w,h); }
   public java.awt.Rectangle getBounds()
   { return path.getBounds(); }
   public Rectangle2D getBounds2D()
   { return path.getBounds2D(); }
   public PathIterator getPathIterator(AffineTransform at)
   { return path.getPathIterator(at); }
   public PathIterator getPathIterator(AffineTransform at, double flatness)
   { return path.getPathIterator(at, flatness); }
   public boolean intersects(Rectangle2D r)
   { return path.intersects(r); }
   public boolean intersects(double x, double y, double w, double h)
   { return path.intersects(x,y,w,h); }
}
```

This is an example of an adapter class since it adapts a `GeneralPath` object by hiding its complexity and provides a simple representation of a specific geometric object such as a triangle or a polygon. Each method is implemented by simply returning what `path`'s version of each method produces.

Now if we have a graphics class such as `MyGraphicsShape` for drawing some geometric object we can write this class in the form

```
public class MyGraphicsShape extends ShapeAdapter
{
   // data fields, if any
   // constructors using the inherited path object to define the path
   // methods, if any
}
```

Since `ShapeAdapter` implements the `Shape` interface our `MyGraphicsShape` class will automatically implement it too.

Now we can use statements such a

```
Shape s = new MyGraphicsShape(....);
...
g2D.draw(s);
```

to construct and draw an object of our class.

9.11.3 `Triangle2D` class that uses `ShapeAdapter`

As an example consider the following `Triangle2D` class that extends `ShapeAdapter` and therefore implements the `Shape` interface.

Class `Triangle2D`

`book-project/chapter9/shapes`

```
package chapter9.shapes;
import java.awt.geom.*;

/**
 * A Triangle2D object is represented as 3 Point objects for
 * the vertices and a path that can be used to draw or fill it.
 * This version of the class also implements the Shape interface
 * by extending the ShapeAdapter class
 */
public class Triangle2D extends ShapeAdapter
{
   private Point2D.Double v1, v2, v3; // the triangle vertices

   /** Construct default triangle with vertices (0,0), (1,0), (0.5,1)
   */
   public Triangle2D()
   {
      this(new Point2D.Double(0,0), new Point2D.Double(1,0),
         new Point2D.Double(0.5,1));
   }

   /**
    * Construct triangle with specified vertices.
    * @param x1 x coordinate of vertex 1
    * @param y1 y coordinate of vertex 1
    * @param x2 x coordinate of vertex 2
    * @param y2 y coordinate of vertex 2
    * @param x3 x coordinate of vertex 3
    * @param y3 y coordinate of vertex 3
    */
   public Triangle2D(double x1, double y1, double x2, double y2, double x3,
      double y3)
   {
      this(new Point2D.Double(x1,y1), new Point2D.Double(x2,y2),
         new Point2D.Double(x3,y3));
   }

   /**
    * Construct triangle with specified points as vertices.
    * @param p1 First vertex
    * @param p2 Second vertex
    * @param p3 Third vertex
    */
   public Triangle2D(Point2D.Double p1, Point2D.Double p2, Point2D.Double p3)
   {
      v1 = (Point2D.Double) p1.clone();
      v2 = (Point2D.Double) p2.clone();
      v3 = (Point2D.Double) p3.clone();
      path.moveTo((float) v1.x, (float) v1.y); // path inherited from ShapeAdapter
      path.lineTo((float) v2.x, (float) v2.y);
      path.lineTo((float) v3.x, (float) v3.y);
      path.closePath();
   }
}
```

Here we have used the `clone` method in the `Point2D.Double` class to make copies of the points specified by the formal arguments so that each `Triangle2D` object will have its own copies of these points as instance data fields.

We can use this class in the following `RandomTriangles` class that draws some random triangles.

Class `RandomTriangles`

`book-project/chapter9/shapes`

```
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;

/**
 * Use the Triangle2D class to draw some random triangles.
 * In this version a Triangle2D object is a Shape.
 */
public class RandomTriangles extends JPanel
{
   private int numTriangles;

   /**
    * Construct object.
    * @param n the number of triangles to draw
    */
   public RandomTriangles(int n)
   {
      setNumTriangles(n);
      setBackground(Color.white); // set panel background color
   }

   public void setNumTriangles(int n)
   {
      numTriangles = n;
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
         RenderingHints.VALUE_ANTIALIAS_ON);

      double w = getWidth();  // panel width in pixels
      double h = getHeight(); // panel height in pixels

      for (int k = 1; k <= numTriangles; k++)
      {
         // Generate random triangle that fits inside the panel

         Triangle2D t = new Triangle2D(
               w*Math.random(), h*Math.random(),
               w*Math.random(), h*Math.random(),
               w*Math.random(), h*Math.random() );

         // RGB values are in the range 0 to 255

         Color c = new Color( (int) (256*Math.random()),
               (int) (256*Math.random()), (int) (256*Math.random()) );

         // Fill triangle with random color and outline it in black

         g2D.setPaint(c);
         g2D.fill(t); // t is now a Shape
         g2D.setPaint(Color.black);
         g2D.draw(t);
      }
   }

   /** draw some random triangles.
    */
   public void draw()
   {
      new GraphicsFrame("Random Triangles",
         new RandomTriangles(numTriangles), 301, 201);
   }

   public static void main(String[] args)
   {
      int n;
      if (args.length == 1)
         n = Integer.parseInt(args[0]);
      else
         n = 10;
      RandomTriangles tri = new RandomTriangles(n);
      tri.draw();
   }
}
```

Note that we do not use `Triangle2D.Double` (we are not providing both `Float` and `Double` versions).

9.11.4 Implementing `Shape` directly

Since a class can only extend one other class we run into a problem if `MyGraphicsShape` is already extending some class. In this case we can implement the `Shape` interface directly. For example,

```
public class MyGraphicsClass extends AnotherClass implements Shape
{
   GeneralPath path;
   // other data fields
   // constructors and methods not in Shape interface
   // implementation of the 10 shape methods go here using path
}
```

9.12 Turtle graphics class

As a final example of implementing the `Shape` interface using `ShapeAdapter` we consider a turtle model for producing line drawings known as "turtle graphics" which originated as part of the Logo computer language.

We think of a turtle moving on a page. The turtle has a pen underneath which can be either up or down. If the pen is down then a line is drawn on the page as the turtle moves. This drawing model is similar to the one used by a pen plotter.

9.12.1 Specification of the class

The turtle can execute the following operations and motions that will correspond to instance methods in a `Turtle2D` class:

- There is a *home* operation that puts the turtle in a default "home" position which we choose to be the origin $(0,0)$ with the turtle pointing north. When a turtle is constructed it is placed in the home position. We will make $(0,0)$ correspond to the center of the drawing panel.
- It can move *forward* along a line a specified distance in the direction it is pointing. This direction is called the turtle heading.
- It can back up (move *backward*) by a specified distance along the line in the opposite direction from which it is pointing without changing its heading.
- It can rotate *left* in place through a specified angle to change its heading. This corresponds to a counterclockwise rotation if the angle is positive.
- It can rotate *right* in place through a specified angle to change its heading. This corresponds to a clockwise rotation if the angle is positive.
- It can retract the pen so that nothing is drawn by the *forward* and *backward* commands.
- It can put its pen down so that drawing takes place.

The `Turtle2D` class will use a `GeneralPath` object `path`, provided by `ShapeAdapter`, to construct the turtle's path. The class specification is

```
public class Turtle2D extends ShapeAdapter
{
   // data fields go here

   // Construct turtle given an approximate bounding rectangle.
   // The home position is at the center of this rectangle.
   public Turtle2D(double xTopLeft, double yTopLeft,
      double width, double height) {...}

   // move the turtle home
   public void home() {...}

   // rotate turtle by given angle counterclockwise
   public void left(double angle) {...}

   // rotate turtle by given angle clockwise
   public void right(double angle) {...}

   // move turtle forward a given distance
   public void forward(double distance) {...}

   // move turtle backward a given distance
   public void backward(double distance) {...}

   // move the pen up
   public void penUp() {...}

   // move the pen down
   public void penDown() {...}
}
```

9.12.2 Implementation of the class

As data fields we need the turtle's position (x,y), the turtle's heading angle, and a boolean variable `penUp` that is true if the pen is in the up position and false otherwise. We also need variables to describe the bounding rectangle of the path produced by the turtle.

```
private double x, y;
private double heading; // in degrees
private boolean penUp;
private double xTopLeft, yTopLeft, width, height;
```

The `path` data field is inherited from `ShapeAdapter`.

To implement the `forward` method we need to use some trigonometry to express the coordinates of a point (x_1,y_1) in terms of polar coordinates as

$$\begin{aligned} x_1 &= r\cos\theta \\ y_1 &= r\sin\theta \end{aligned}$$

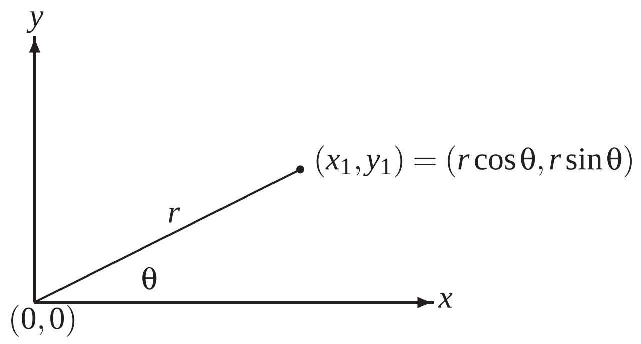


Figure 9.11: Polar coordinates of a point (x_1,y_1)

where r is the distance from (x_1,y_1) to the origin $(0,0)$ and θ is the angle in radians measured from the x-axis as shown in Figure 9.11. Since the turtle is at position (x,y), not $(0,0)$ we can translate the coordinates to obtain

$$\begin{aligned} x_1 &= x+r\cos\theta \\ y_1 &= y+r\sin\theta \end{aligned}$$

as the new coordinates of the turtle corresponding to *forward(r)*. Since (x_1,y_1) becomes the new (x,y) we can write

$$\begin{aligned} x &\leftarrow x+r\cos\theta \\ y &\leftarrow y+r\sin\theta \end{aligned}$$

Since we are dealing with a user space in Java 2D that has origin at the top left corner with the y-axis pointing downward, we need to change to a minus sign on the right side of the equation for y. This gives the method implementation

```
public void forward(double distance)
{
   double radians = Math.toRadians(heading);
   x = x + distance * Math.cos(radians);
   y = y - distance * Math.sin(radians);
   if (penUp)
      path.moveTo((float)x, (float)y);
   else
      path.lineTo((float)x, (float)y);
}
```

We can call this method to obtain `backward`:

```
public void backward(double distance)
{
   forward(-distance);
}
```

We can now implement `left` and `right`. Our first attempt at `left` would be

```
public void left(double angle)
{
   heading = heading + angle;
}
```

However if `left` is called many times the angle can become quite large and the `sin` and `cos` functions will produce more round-off error. Therefore it is useful to always adjust the heading so that it is between 0 degrees and 360 degrees each time `left` is called. This gives the implementation

```
public void left(double angle)
{
   heading = heading + angle
   while (heading > 360.0) heading = heading - 360.0;
   while (heading < 0.0) heading = heading + 360.0;
}
```

and similarly for `right`, where the angle is subtracted from the heading.

Here is the complete class declaration that also includes two methods called `setHeading` and `setDirection` for specifying an absolute rather than relative heading and direction.

Class `Turtle2D`

book-project/chapter9/shapes

```
package chapter9.shapes;

/**
 * A class that defines Turtle objects for drawing on a panel.
 * Each turtle object produces a Shape object which can be drawn
 * using the draw method of a Graphics2D graphics context.
 * <p>
 * The approximate bounding rectangle for the turtle's path is
 * specified in the constructor but the final rectangle will generally
 * be different and can be obtained using the getBounds2D
 * method of the Shape interface.
 */
public class Turtle2D extends ShapeAdapter
{
   private double x, y;           // coordinates of turtle
   private double heading;        // turtle direction in degrees
   private boolean penUp;         // turtle has a pen to draw with

   // Definition of the bounding rectangle
   private double xTopLeft, yTopLeft, width, height;

   /**
    * Construct a turtle that will draw a path.
    * The turtle state is defined by its current position and heading
    * in degrees, and by its pen state (up or down).
    * <p>
    * The turtle begins at (0,0) facing north with its pen down.
    * @param xTopLeft top left x-coord of bounding rectangle
    * @param yTopLeft top left y-coord of bounding rectangle
    * @param width width of the bounding rectangle
    * @param height of the bounding rectangle
    */
   public Turtle2D(double xTopLeft, double yTopLeft,
      double width, double height)
   {
      penUp = false;
      this.xTopLeft = xTopLeft;
      this.yTopLeft = yTopLeft;
      this.width = width;
      this.height = height;
      home();
   }

   /**
    * Move turtle to center, facing north
    */
   public void home()
   {
      x = xTopLeft + width / 2.0;    // x coord of center
      y = yTopLeft + height / 2.0;   // y coord of center
      heading = 90.0;                       // north default
      path.moveTo((float)x,(float)y);       // set the position
   }

   /**
    * Set the turtle at a global position without changing its heading.
    * @param x x-coord of the position
    * @param y y-coord of the position
    */
   public void setPosition(double x, double y)
   {
      this.x = x;
      this.y = y;
      path.moveTo((float)x,(float)y);
   }

   /**
    * Set the turtle to a specified heading without changing its position.
    * @param heading angle in degrees (0 = east, 90 = north)
    */
   public void setHeading(double heading)
   {
      this.heading = heading;
   }

   /**
    * Rotate the turtle counterclockwise.
```

```
    * @param angle rotation angle in degrees
    */
   public void left(double angle)
   {
      heading = heading + angle;
      while (heading > 360.0) heading = heading - 360.0;
      while (heading < 0.0) heading = heading + 360.0;
   }


   /**
    * Rotate the turtle clockwise.
    * @param angle rotation angle in degrees
    */
   public void right(double angle)
   {
      heading = heading - angle;
      while (heading > 360.0) heading = heading - 360.0;
      while (heading < 0.0) heading = heading + 360.0;
   }

   /**
    * Advance turtle a given distance along its heading.
    * @param distance the distance to advance. A negative
    * value would move the turtle backward.
    */
   public void forward(double distance)
   {
      double radians = Math.toRadians(heading);

      x = x + distance * Math.cos(radians);

      // negative sign in y since we have a user space with
      // origin at top left corner of panel and a y-axis
      // pointing downward.

      y = y - distance * Math.sin(radians);

      if (penUp)
         path.moveTo((float)x, (float)y);
      else
         path.lineTo((float)x, (float)y);
   }

   /**
    * Advance turtle a given distance backward from its heading.
    * @param distance the distance to move backward. A negative
    * value would move the turtle forward.
    */
   public void backward(double distance)
   {
      forward(-distance);
   }

   /**
    * Set the turtle’s pen in the up position.
    * In this state forward and backward move turtle without drawing.
    */
   public void penUp()
   {
      penUp = true;
   }

   /**
    * Set the turtle’s pen in the down position.
    * In this state forward and backward move turtle with drawing.
    */
   public void penDown()
   {
      penUp = false;
   }
}
```

9.12.3 Writing turtle graphics programs

We can use the `GraphicsFrame` class to write turtle graphics programs. Here is a simple template:

```
import custom_classes.GraphicsFrame;
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class MyTurtleGraph extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;

      // Construct a turtle for the entire drawing panel
      Turtle2D t = new Turtle2D(0,0, getWidth(), getHeight());

      // insert turtle graphics commands here to construct turtle path

      g2D.draw(t); // draw the turtle’s path
   }

   public static void main(String[] args)
   {
      new GraphicsFrame("MyTurtleGraph", new MyTurtleGraph(), 301, 301);
   }
```

```
      }
```

■ EXAMPLE 9.10 **(Drawing a box)** Since the turtle starts at $(0,0)$ pointing north, the statements

```
    for (int k = 1; k <= 2; k++)
    {
       t.forward(10);
       t.right(90);
       t.forward(20);
       t.right(90);
    }
```

define a rectangle with lower left corner at $(0,0)$ and upper right corner at $(20,10)$. ■

■ EXAMPLE 9.11 **(Drawing a pentagon)** The statements

```
    for (int k = 1; k <= 5; k++)
    {
       t.forward(80);
       t.right(72);
    }
```

show how easy it is to draw a pentagon with a vertex at $(0,0)$. ■

Here is a class called `PentagonSpinner` that draws pentagons rotated about the lower left corner. Each time the pentagon is drawn the turtle rotates 36 degrees before drawing the next one.

Class `PentagonSpinner`

`book-project/chapter9/shapes`

```
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;

/**
 * Use A Turtle2D object to draw a pentagon and spin it.
 */
public class PentagonSpinner extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setPaint(Color.blue);
      g2D.setStroke(new BasicStroke(2.0f));
      Turtle2D t = new Turtle2D(0,0, getWidth(), getHeight());

      for (int k = 1; k <= 10; k++)
      {
         drawPentagon(t);
         t.left(36);
      }

      g2D.draw(t);
   }

   public void drawPentagon(Turtle2D t)
   {
      for (int k = 1; k <= 5; k++)
      {
         t.forward(80);
         t.right(72);
      }
   }

   public void draw()
   {
      new GraphicsFrame("Spinning Pentagons", this, 301, 301);
   }

   public static void main(String[] args)
   {
      new PentagonSpinner().draw();
   }
}
```

The output is shown in Figure 9.12. The turtle graphics model makes it easy to draw pictures like this.

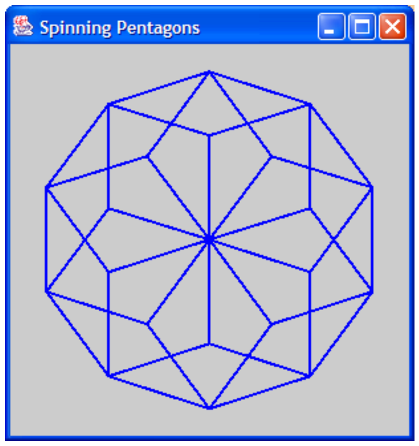


Figure 9.12: Output for `PentagonSpinner` program.

9.12.4 Recursive turtle graphics programs

Many interesting pictures can be drawn using recursive turtle graphics methods. Here is a method that draws the branches of a tree.

```
public void tree(double length)
{
   if (length < 1) return;

   t.right(45);
   t.forward(length);
   tree(length / 1.7);
   t.backward(length);
   t.left(90);
   t.forward(length);
   tree(length / 1.7);
   t.backward(length);
   t.right(45);
}
```

The turtle is pointing north by default so it first rotates right by 45 degrees and draws the right branches, then it backs up and rotates left by 90 degrees and draws the left branches. Finally, it rotates right by 45 degrees so it is pointing north again. The recursion is controlled by the length of the branches. At each recursive step the branch length decreases and the recursion is stopped if the length becomes less than 1. Here is a class that uses this method.

Class `RecursiveTreeMaker`

book-project/chapter9/shapes

```
package chapter9.shapes;
import custom_classes.GraphicsFrame;
import java.awt.*;
import javax.swing.*;

/**
 * Use a Turtle2D object to draw a recursive tree.
 */
public class RecursiveTreeMaker extends JPanel
{
   Turtle2D t; // reference to the turtle

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2D = (Graphics2D) g;
      g2D.setColor(Color.blue);
      g2D.setStroke(new BasicStroke(2.0f));
      t = new Turtle2D(0,0, getWidth(), getHeight());
      t.backward(100);  // make the tree trunk
      t.forward(100);
      tree(50);         // make tree branches
      g2D.draw(t);
   }

   /**
    * Recursive method for drawing the branches of a tree.
    * At each step the branch length decreases.
    * When it gets small enough the recursion is stopped.
    */
   public void tree(double length)
   {
      if (length < 1) return;

      t.right(45);          // position turtle for right side

      t.forward(length);    // draw right branches
      tree(length / 1.7);
      t.backward(length);

      t.left(90);           // position turtle for left side

      t.forward(length);    // draw left branches
      tree(length / 1.7);
      t.backward(length);

      t.right(45);          // position turtle for right side
   }

   public void draw()
   {
      new GraphicsFrame("Recursive Tree", this, 225, 225);
   }

   public static void main(String[] args)
   {
      new RecursiveTreeMaker().draw();
   }
}
```

The output window is shown in Figure 9.13.

9.13 Numerical applications of interfaces

In numerical analysis it is quite common to work with many kinds of functions and it is necessary to write algorithms that have functions as their input. In Java this means that we need a way to use a function as a method argument. This can be done in an elegant fashion using interfaces. We illustrate this with two examples: (1) a class containing methods that display a table of values of a function and (2) a class containing methods to do function iteration.

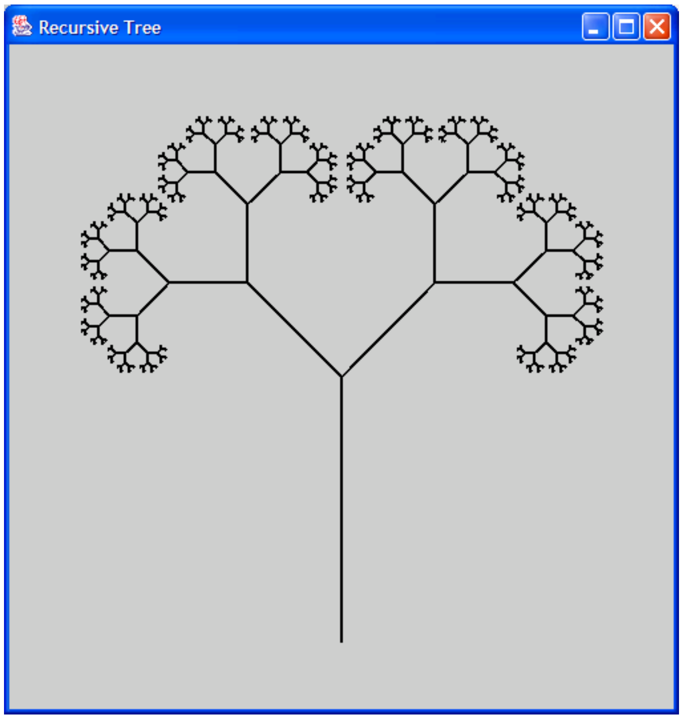


Figure 9.13: Output window for the RecursiveTreeMaker class

9.13.1 Displaying a table of values of a function

Suppose we have a function f with values $f(x)$ and we want to display a table of its values at the points $x_{start}, x_{start+1}, \ldots, x_{end}$. Assume that the x values are equally spaced so they have the form x_{start}, $x_{start} + h$, $x_{start} + 2h$, $\ldots$, x_{end}, where h is the distance between successive x values. Given x_{start} and x_{end} the x values for the table can be expressed as $x_k = x_{start} + kh, k = 0, \ldots, n$ and $x_{end} = x_{start} + nh$, where n is the number of steps and $n+1$ is the number of table values. A simple method for computing a table of values is

```
public void table(double xStart, double xEnd, double h)
{
   int numSteps = (int) Math.round((xEnd - xStart) / h);
   for (int k = 0; k <= numSteps; k++)
   {
      double x = xStart + k*h;
      System.out.println(x + " " + f(x));
   }
}
```

We also need a method to define the function f. For example, if $f(x) = e^{-x}$ then we can write the method

```
public double f(double x)
{
   return Math.exp(-x);
}
```

to return values of this function.

What happens if we want to produce a table for another function? We run into a problem since the function name `f` is built-in to our `table` method. We would have to change the body of the function `f` and re-compile the class containing it. Thus our table method is not reusable because the function is not supplied as an argument.

The solution is to write a function as an object from a class that implements a function interface. For example, we can use the following interface to represent a double-valued function of a double variable.

Interface `DoubleFunction`

`book-project/chapter9/functions`

```
package chapter9.functions;
public interface DoubleFunction
{
   /**
    * Return value of a function f(x) at a given x.
    * @param x value at which to evaluate the function
    * @return the value f(x) of the function
    */
   public double value(double x);
}
```

This interface represents all functions that have one `double` argument and return one `double` value. Any class that implements this interface must provide an implementation of the `value` method that gives the value of the function. An object of this class 'is a' `DoubleFunction`. Now we can write the table method in a class called `TableMaker` using a `DoubleFunction` argument as

Class `TableMaker`

`book-project/chapter9/functions`

```
package chapter9.functions;
public class TableMaker
{
   private String fmt; // format code

   /**
    * Construct a table maker with formatting
    * @param fmt the format code for printing
    */
   public TableMaker(String fmt)
   {
      this.fmt = fmt;
   }

   /**
    * Display a table of the function f.
    * @param f the function
    * @param xStart starting value of x
    * @param xEnd ending value of x
    * @param h step size between teble entries
    */
   public void table(DoubleFunction f, double xStart, double xEnd, double h)
   {
      int numSteps = (int) Math.round((xEnd - xStart) / h);
      for (int k = 0; k <= numSteps; k++)
      {
         double x = xStart + k*h;
         System.out.printf(fmt, x, f.value(x));
      }
   }
}
```

Here we have included a simple formatting code for each line of the table. For example

```
TableMaker maker = new TableMaker("%5.2f %10.5f\n");
```

The important idea here is that the first argument of the `table` method can be an object of any class that implements the `DoubleFunction` interface so this `table` method is reusable. We simply use `f.value(x)` in the method body to obtain the value of the function at `x`.

For example, the functions $f(x) = e^{-x}$ and $g(x) = \cos x$ can be represented as objects of the following two simple classes that implement the `DoubleFunction` interface.

Class `ExpMinusFunction`

book-project/chapter9/functions

```
package chapter9.functions;
/**
 * Class for the function exp(-x)
 */
public class ExpMinusFunction implements DoubleFunction
{
   public double value(double x)
   {
      return Math.exp(-x);
   }
}
```

Class `CosFunction`

book-project/chapter9/functions

```
package chapter9.functions;
/**
 * Class for the function cos(x)
 */
public class CosFunction implements DoubleFunction
{
   public double value(double x)
   {
      return Math.cos(x);
   }
}
```

The following tester class shows how to produce tables for these two functions.

Class `TableMakerTester`

`book-project/chapter9/functions`

```
package chapter9.functions;
public class TableMakerTester
{
   public void doTest()
   {
      TableMaker maker = new TableMaker("%5.2f %10.5f\n");
      DoubleFunction exp = new ExpMinusFunction();
      DoubleFunction cos = new CosFunction();

      System.out.println("Table of exp(-x)");
      maker.table(exp, 0.0, 0.5, 0.1);
      System.out.println("Table of cos(x)");
      maker.table(cos, 0.0, 0.5, 0.1);
   }

   public static void main(String[] args)
   {
      TableMakerTester tester = new TableMakerTester();
      tester.doTest();
   }
}
```

The output is

```
Table of exp(-x)
 0.00    1.00000
 0.10    0.90484
 0.20    0.81873
 0.30    0.74082
 0.40    0.67032
 0.50    0.60653
Table of cos(x)
 0.00    1.00000
 0.10    0.99500
 0.20    0.98007
 0.30    0.95534
 0.40    0.92106
 0.50    0.87758
```

9.13.2 Function iteration

The iterates of a function $f(x)$ form a sequence $x_0, x_1, \ldots, x_n, \ldots$, defined by $x_k = f(x_{k-1})$. Here we assume that x_0 is given. We also require that all the values x_k are defined (they are in the domain of definition of the function f). We are interested in the long-term behaviour of the sequence for a given function f and initial value x_0. There are three possibilities: (1) the sequence converges, (2) the sequence diverges, (3) the sequence neither converges nor diverges. The third case can be quite interesting.

For example, if you choose the function $f(x) = 0.5(x+2/x)$ and iterate it starting with $x_0 = 1$ then the sequence converges to $\sqrt{2}$:

$$x_0 = 1,\ x_1 = 0.5(x_0+2/x_0) = 1.5,\ x_2 = 0.5(x_1+2/x_1) = 1.41666...,\ldots$$

If $f(x) = 2x$ then the sequence clearly diverges.

An interesting case illustrating the third possibility is $f(x) = 3.83x(1-x)$. Here the sequence of iterates neither converges nor diverges. Instead it eventually repeats in a periodic manner in groups of 3. For example the sequence eventually looks like

$$\begin{gathered}0.9574165975188731,\ 0.15614931568360532,\ 0.5046664874084134,\\ 0.9574165975188731,\ 0.15614931568360532,\ 0.5046664874084134,\\ 0.9574165975188731,\ 0.15614931568360532,\ 0.5046664874084134,\ldots\end{gathered}$$

Here is a class containing two methods `iterate` and `iterate2` that can be used to iterate a function. The `iterate` method computes and displays the sequence $x_0, x_1, \ldots, x_{n-1}$ and `iterate2` lets you skip iterations $x_0, x_1, \ldots x_{\text{skip}-1}$ before displaying the sequence $x_{\text{skip}}, x_{\text{skip}+1}, \ldots x_{\text{skip}+n-1}$. Each method has a `DoubleFunction` argument to specify the function to iterate.

Class `FunctionIterator`

book-project/chapter9/functions

```
package chapter9.functions;
/**
 * This class shows how to iterate a function f(x)
 * using the DoubleFunction interface.
 */
public class FunctionIterator
{
   /**
    * Compute iterates of a function.
    * @param f the function to iterate
    * @param x0 the initial value of x
    * @param n number of iterations to display
    * The values x(0),...,x(n-1) are displayed
    */
   public void iterate(DoubleFunction f, double x0, int n)
   {
      double x = x0;
      System.out.println(x);
      for (int k = 1; k < n; k++)
      {
         x = f.value(x);
         System.out.println(x);
      }
   }
   /**
    * Compute iterates of a function, skiping the first few.
    * @param f the function to iterate
    * @param x0 the initial value of x
    * @param skip the number of iteration to skip before display
    * @param n the number of iterations to display
    * The values x(0),...,x(skip-1) are skipped, then the values
    * x(skip), ..., x(skip+n-1) are displayed.
    */
   public void iterate2(DoubleFunction f, double x0, int skip, int n)
   {
      double x = x0;
      for (int k = 0; k < skip; k++) x = f.value(x);
      iterate(f, x, n);
   }
}
```

Here is a tester class that shows how to iterate the functions $0.5(x+2/x)$ and $3.83x(1-x)$ using inner classes to define the functions.

Class `FunctionIteratorTester`

`book-project/chapter9/functions`

```
package chapter9.functions;
public class FunctionIteratorTester
{
   public void doTest()
   {
      FunctionIterator f = new FunctionIterator();
      DoubleFunction sqrt = new Sqrt();

      System.out.println("5 Iterates of sqrt");
      f.iterate(sqrt, 1.0, 5);

      // Test iterate2 with a period 3 function
      // skip 10000 iterates and display 10

      DoubleFunction period3 = new Period3();
      System.out.println("Skip 10000 iterates of 3.83*x*(1-x)");
      f.iterate2(period3, 0.1, 10000, 6);
   }

   private static class Sqrt implements DoubleFunction
   {
      public double value(double x)
      {
         return 0.5*(x + 2/x);
      }
   }
   private static class Period3 implements DoubleFunction
   {
      public double value(double x)
      {
         return 3.83*x*(1-x);
      }
   }

   public static void main(String[] args)
   {
      new FunctionIteratorTester().doTest();
   }
}
```

The output is

```
5 Iterates of sqrt
1.0
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
Skip 10000 iterates of 3.83*x*(1-x)
0.9574165975188731
0.15614931568360532
0.5046664874084134
0.9574165975188731
0.15614931568360532
0.5046664874084134
```

Sometimes functions contain parameters. For example, we can find the square root of the number $a \geq 0$ by iterating the function $f_a(x) = 0.5(x + a/x)$ and we can generalize the period 3 example to $f_a(x) = ax(1-x)$. The following example shows how to do the parametrized square root function

Class `SquareRootIterator`

`book-project/chapter9/functions`

```
package chapter9.functions;
public class SquareRootIterator
{
   public void doTest()
   {
      FunctionIterator f = new FunctionIterator();
      DoubleFunction sqrt = new Sqrt(3.0);
      f.iterate(sqrt, 1.0, 7);
   }

   private static class Sqrt implements DoubleFunction
   {
      private double a;

      public Sqrt(double a)
      {
         this.a = a;
      }

      public double value(double x)
      {
         return 0.5*(x + a/x);
      }
   }

   public static void main(String[] args)
   {
      new SquareRootIterator().doTest();
   }
}
```

The output for $\sqrt{3}$ is

```
1.0
2.0
1.75
1.7321428571428572
1.7320508100147274
1.7320508075688772
1.7320508075688772
```

The trick is to include a constructor in the `Sqrt` class that sets the value of the private data field `a` defining the parameter. This parameter is set using

```
DoubleFunction sqrt = new Sqrt(3.0);
```

9.14 Review exercises

► **Review Exercise 9.1** Define the following terms and give examples of each.

inheritance	polymorphism	polymorphic type
polymorphic method	adapter class	abstract class
abstract method	interface	multiple inheritance
subclass	superclass	`Object` class
`extends`	`this(...)`	`super`
`protected`	`toString`	`Shape` interface
`equals`	base class	`final` class

► **Review Exercise 9.2** Write a tester class called `Amnesia` similar to `CircleCalculatorTester` on page 459 but with the following test method.

```
public void test()
{
   CircleCalculatorA circle = new CircleCalculatorB(3.0);
   double circumference = circle.getCircumference();
   System.out.println("Circumference: " + circumference);
}
```

Explain what happens? Replace the line defining the circumference by

```
double circumference = ((CircleCalculatorB)circle).getCircumference();
```

and explain why this works.

9.15 Programming exercises

► Exercise 9.1 (Inheritance using TriangleCalculator)
In Chapter 3 we considered a class called `CircleCalculator` and in Chapter 9 we wrote the class `CircleCalculatorA` which calculates only the area and then we extended this class to `CircleCalculatorB` which calculated the circumference.

Do something similar with the `TriangleCalculator` class from Chapter 3, page 64 as follows:

- Write a class called `TriangleCalculatorA` that computes only the third side length `c` and the three angles `alpha`, `beta` and `gamma`. Also include a `toString` method.
- Write a class called `TriangleCalculatorB` that extends `TriangleCalculatorA` by additionally calculating the perimeter and the area.
- Write a tester class called `TriangleCalculatorTester` that tests the two classes.

► Exercise 9.2 (Polymorphic bank account transfer method)
Write a tester class for the polymorphic bank account `transfer` method (see page 469).

► Exercise 9.3 (Completing the Employee class hierarchy)
Complete the remaining three classes in the `Employee` hierarchy and test the class hierarchy using `EmployeeProcessor`, page 474.

► Exercise 9.4 (Completing the Employee class hierarchy)
Remove the `toString` methods from the four classes in the `Employee` hierarchy and put the following `toString` method into the `Employee` class.

```
public String toString()
{
   return getClass().getName() + "[" + "name = " + getName() +
      ", gross = " + grossSalary() + ", net = " + netSalary() + "]";
}
```

Compile the class and use `EmployeeProcessor`, page 474 to test it. Can you explain why this `toString` method works.

► Exercise 9.5 (Another way to design the Employee hierarchy)
Sometimes there is more than one way to design a hierarchy. For example, in the `Employee` hierarchy we could say that a part time worker is an hourly worker (one without deductions). This gives the hierarchy shown in Figure 9.14. Modify the classes of the preceding exercise to use this hierarchy.

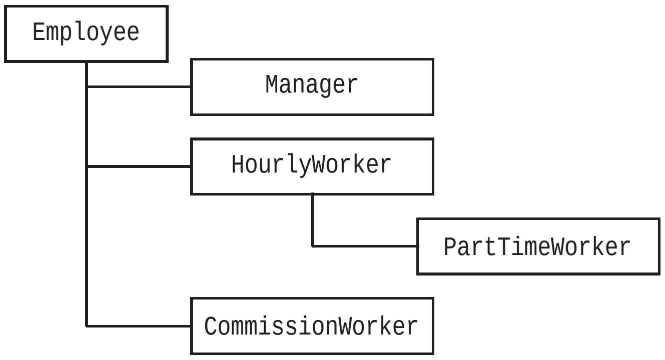


Figure 9.14: Another possible Employee inheritance hierarchy

► **Exercise 9.6 (A student employee hierarchy)**
Write a class called `Person` and three subclasses called `Student` (a subclass of `Person`), `Employee` (a subclass of `Person`) and `StudentEmployee` (a subclass of `Student`) as follows

(a) The `Person` class encapsulates three private data fields: a `name` of type `String`, a social insurance number of type `String` and a year of birth of type `int`. The class should include a constructor, get methods for the private data fields and a `toString` method.

(b) The `Student` class is a subclass of `Person`. A student is a person with a major (MATH, COSC, etc.) and a student number. Provide a constructor, the appropriate get methods and a `toString` method.

(c) The `Employee` class is a subclass of `Person`. An employee is a person with a monthly salary. Provide a constructor, the appropriate get methods and a `toString` method.

(d) The `StudentEmployee` class is a subclass of `Student`. A student employee is a student with a salary. Provide a constructor, the appropriate get method and a `toString` method.

► **Exercise 9.7 (Equals method for bank account classes)**
Write the `equals` method for the `BankAccount` and `JointBankAccount` classes assuming that two accounts are equal if they have the same account number.

► **Exercise 9.8 (Using the Translatable interface)**
Write a class called `TranslatableTester`, similar to `MeasurableTester`, that declares an array of type `Translatable`. Write the for-loop so that it translates each object by (1,1) and displays results.

► **Exercise 9.9 (Using the Scalable interface)**
Write a class called `ScalableTester`, similar to `MeasurableTester`, that declares an array of type `Scalable`. Write the for-loop so that it scales each object by a factor of 2 and displays results.

► **Exercise 9.10 (An Employable interface)**
Do the `Employee` hierarchy in Section 9.6 using an `Employable` interface instead of an inheritance hierarchy. This interface is defined by

```
public interface Employable
{
   /**
    * Return the gross monthly salary.
    * @return the gross monthly salary
    */
   public double grossSalaray();
   /**
    * Return the net monthly salary.
    * @return the net monthly salary
    */
   public double netSalary();
}
```

Now there is no abstract `Employee` class. Instead, each of the classes `Manager`, `HourlyWorker`, `PartTimeWorker`, and `CommissionWorker` will need to implement the `Employable` interface. Each class must now have a `name` field and a `getName` method since we no longer have the `Employee` class to manage the name for us.

Modify the tester class `EmployeeProcessor` to use an array of `Employable` references instead of an array of `Employee` references. We now have an array of references to objects from classes that implement the `Employable` interface instead of an array of references to objects that extend the abstract `Employee` class.

Discuss any advantages or disadvantages of these two approaches to polymorphism.

► Exercise 9.11 (An equals method for the Circle class)
Write an `equals` method for the `Circle` class from Chapter 4.4.5, page 122

► Exercise 9.12 (Colored circles)
Consider the following `Point` and `Circle` and `Color` classes:

```
public class Point
{  private double x, y;
   public Point() {...}
   public Point(double x, double y) {...}
   public double getX() {...}
   public double getY() {...}
   public String toString() {...}
}

public class Circle
{  private Point center;
   private double radius;
   public Circle() {...}
   public Circle(double x, double y, double radius) {...}
   public Circle(Point center, double radius) {...}
   public Point getCenter() {...}
   public double getRadius() {...}
   public String toString() {...}
}

public class Color
{  private int red, green, blue;
   public Color() {...}
   public Color(int red, int green, int blue) {...}
   public int getRed() {...}
   public int getGreen() {...}
   public int getBlue() {...}
   public String toString() {...}
}
```

Write a class called `ColoredCircle` that extends `Circle` by adding a new data field for the circle color and has constructors

```
public ColoredCircle() {...}
public ColoredCircle(double x, double y, double radius, Color color) {...}
public ColoredCircle(Circle circle, Color color) {...}
public ColoredCircle(Point center, double radius, Color color {...}
```

Assuming that `Color` and `Circle` are immutable classes (objects cannot be modified after construction) write the `ColoredCircle` class so that it is also immutable.

► Exercise 9.13 (A polygon Shape class)
Write a `Polygon2D` class that extends the `ShapeAdapter` class (it implements the `Shape` interface). Rewrite the `RandomPolygon` program class (page 497) so that it uses the `Polygon2D` class.

► Exercise 9.14 (A pentagon Shape class)
Write a `Pentagon2D` class that extends the `ShapeAdapter` class and defines regular pentagons (5 sided polygons with all sides equal and central angle between consecutive vertices of 72 degrees). Each pentagon should be defined in user space by the coordinates of its center and the radius of the circumscribed circle, rather then the coordinates of its five vertices. Include a constructor of the form

```
public Pentagon2D(double xCenter, double yCenter, double radius,
   double angle)
```

where `angle` is the angle in degrees of one of the vertices. See Chapter 8, Exercise 8.8 for the definitions of the pentagonal angles and vertices. Write a test program using `GraphicsFrame` that draws some pentagons.

► Exercise 9.15 (A happy face Shape class)
Write a `HappyFace2D` class based on the happy face programs developed in Chapter 5 that extends the `ShapeAdapter` class. Use the bounding box to define the size of the face. Include some flexibility in the constructors that lets the user define the face using either the bounding rectangle or the center and radius. Also let the user choose the colors. Write a test program using `GraphicsFrame` that draws some happy faces.

► Exercise 9.16 (A house Shape class)
Write a `House2D` class that extends the `ShapeAdapter` class. A house has a frame, door, windows and chimney. Include some flexibility in the constructors that lets the user define the house using the bounding rectangle and choose the fill colors, for example. Write a test program using `GraphicsFrame` that draws some houses.

► Exercise 9.17 (Polymorphism in the Shape hierarchy)
Write a class that uses the `Shape` objects `Pentagon2D`, `HappyFace2D`, `House2D`, and `Polygon2D` from the previous exercises to illustrate polymorphism. To do this define an array of `Shape` objects, store some of these objects in the array, and use the `draw` method in a loop to draw a picture.

► Exercise 9.18 (A general polygon spinner program)
The `PentagonSpinner` program only spins pentagons and only 10 times using an angle of 36 degrees (10 times 36 is a full circle). Write a `PolygonSpinner` program based on this class that spins a regular polygon. The program should read the number of sides and the spinning angle as command line arguments.

► Exercise 9.19 (Modifying RecursiveTreeMaker)
The `RecursiveTreeMaker` class uses fixed values of 45 and 90 degrees for the angles and 1.7 for the branch length reduction factor. Rewrite the class so that it reads these three values as command line arguments.

► Exercise 9.20 (Function iteration)
Using a class similar to `SquareRootIterator` write a class called `LogisticIterator` that can be used to iterate the function $f_a(x) = ax(1-x)$. Experiment with the long-term beaviour of the iterations for various values of s in the range $0 \le a \le 4$. We have already considered the case $a = 3.83$.

► Exercise 9.21 (Newton's method for root finding)
If $f(x)$ is differentiable with derivative denoted by $f'(x)$ then a root of $f(x) = 0$ can often be calculated using the iteration scheme

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0, 1, \ldots$$

starting with a good initial guess x_0.

Write a class called `RootFinder` that contains a method called `newton` that calculates and displays the iterations. This method will need two `DoubleFunction` arguments, one for $f(x)$ and another for the derivative $f'(x)$. Other arguments can be used to specify the intial guess and the maximum number of iterations. Also specify a tolerance ε such that the iteration process is stopped when either the maximum number of iterations is reached or $|x_{n+1} - x_n| \le \varepsilon$.

Write a tester class that illustrates several examples.

Chapter 10

Graphical Interface Design

Graphical applications and applets

Outline

Basic structure of a GUI application

GUI components and the greeting application

Inner classes as event handlers

Numeric fields and the temperature application

Multi-line text fields (`TextArea`)

Investment application

Using inheritance to design smarter text fields

GUI for the `LoanRepaymentTable` class

Unit conversion application

Inheritance and listener interfaces

Average mark calculator

GUI version of the `RandomTriangles` class

Inheritance and the `GraphicsFrame` class

Applets

10.1 Introduction

There are three kinds of Java programs: Console applications, GUI applications, and applets. So far we have written only console applications. They are programs that use the console window as the user interface. All input is obtained either from command-line arguments or using the `Scanner` class. Even our graphics applications beginning in Chapter 5 were of this form.

In this chapter we introduce graphical user interface (GUI) concepts and design and we write some GUI applications. They run in the window environment and have buttons to click, boxes in which to type, boxes in which to display textual output, and other components as well, such as panels containing graphics. We will also see that a GUI is an event-driven environment whose programming model is quite different from what we are used to with console applications. In order for our programs to respond to events such as pressing the enter key in an input box, clicking a button, clicking or dragging the mouse, it is necessary to write special methods, defined by event listener interfaces, that the system will call when these events occur.

Applets are the third kind of Java program. They always have a GUI but an applet is a special class that is designed to be run by a Web browser. Most of what we learn about GUI applications is also valid for applets so we briefly consider some applet examples and show how they can be tested using a special applet viewer program before running them with a web browser.

10.2 Basic structure of a GUI application

A GUI application class is a subclass of the `JFrame` class. We will call such a class a GUI application. A `main` method is used to construct an application object. The `GraphicsFrame` class used in Chapter 5 is an example. A `JFrame` object is a top-level component that appears as a window with a frame around it containing a title bar at the top. There are also buttons to minimize, maximize, or close the frame. By default the interior of the frame is empty. We will learn how to place other components inside the frame that respond to user actions such as pressing the Enter key after some text has been typed in a box or using the mouse to click a button.

10.2.1 Basic template for GUI applications

Here is a class template that we will use for our simple GUI applications:

Class `ApplicationTemplate`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Use this class as a template for simple GUI applications.
 */
public class ApplicationTemplate extends JFrame implements ActionListener
{
   // Declare instance variables for GUI components here
   // Declare any other instance or class variables here

   public ApplicationTemplate()
   {
      setTitle("Put your frame title here");

      // Statements to create GUI components go here
      // Statements to add them to the frame go here
      // statements to add action listeners go here

      setSize(400,300); // size of the frame
   }

   /**
    * This method is called when an action event occurs.
    * @param e the action event that occurred
    */
   public void actionPerformed(ActionEvent e)
   {
      // statements to process action events go here
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new ApplicationTemplate();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

This is not the most general form of a GUI application class but it will suffice for many of the examples in this chapter. For now, let us give a brief overview of each part of the template:

- There are several `import` statements that are needed to access the GUI classes and the Swing user interface components (the new platform-independent GUI components).
- The class extends `JFrame` which is a top-level component that provides the basic functionality of a window with a frame around it, a title bar with a title in it, and minimize, maximize and close buttons. It is a container for other GUI components that implements the `ActionListener` interface to provide for responses to user interaction. We will also consider approaches to event handling using other classes to implement listener interfaces instead of the application class itself.
- The data field variables are declared at the top of the class. They can be instance variables for the names of GUI component objects, other instance variables, or class variables and constants.

- The `main` method is responsible for constructing an application object as a `JFrame` and the `setVisible` statement makes it visible. Without this statement the frame would remain invisible when the application is running. Also we specify that the application should exit whenever the close button is clicked.

- The constructor is responsible for initializing the application by creating the GUI components and initializing other data field variables. It can also provide a title for the frame using the `setTitle` method and provide the width and height in pixels of the frame using the `setSize` method. The constructor is also be responsible for adding the various event listeners to the appropriate components.

- Finally, we need to write what is called "event processing code". In the simplest cases, when a button is clicked, or the Enter key is pressed while typing text in a box, the component calls an `actionPerformed` method. Our event processing code will go in the body of this method.

The `ApplicationTemplate` class can be compiled and run as usual with the `javac` and `java` commands or from `BlueJ` by right clicking on the class rectangle and selecting the `main` method. When the interpreter executes the statements in the `main` method the frame is created and made visible. Of course this application doesn't do anything so you will see only an empty 400 by 300 pixel frame. Clicking the close box closes the window and terminates the program. To make the GUI application useful we need to learn how to put some graphical user interface components in the window and have them respond to user actions.

10.3 GUI components and the greeting application

10.3.1 Greeting application design

The Java GUI is object-oriented. This means that each GUI **component** you see on the screen is an object from some class (these components are sometime called widgets). The simplest GUI components are objects from the `JLabel`, `JButton`, `JTextField`, and `JTextArea` classes. These classes all have `JComponent` as one of their superclasses.

Our first application will be a simple class called `Greeting1` that asks the user to enter a name in a text field and press the Enter key when done. Then a greeting is displayed in another text field. The greeting is the word `Hello` followed by the name typed. Figure 10.1 shows what the application looks like when the name has been typed. After the Enter key is pressed the application is shown in Figure 10.2. You can easily identify three component objects in the frame: (1) a prompt string, (2) an input box for typing the name, and (3) an output box for displaying the greeting. The boxes are called text fields.

10.3.2 Determining what GUI components are needed

First we need to decide which GUI components are needed. We will use the following three components

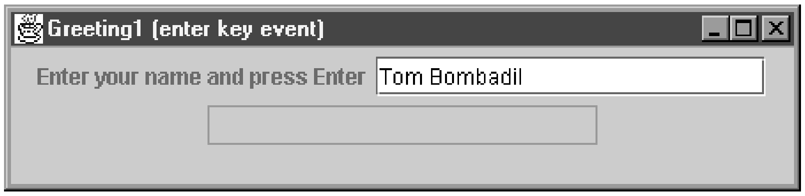


Figure 10.1: Greeting application before Enter key is pressed

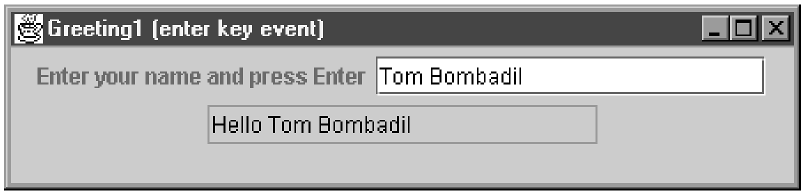


Figure 10.2: Greeting application after Enter key is pressed

- A `JTextField` object called `input` which defines the field in which the name can be typed (white box in Figure 10.1 or Figure 10.2).
- A `JLabel` object called `prompt` which provides a place on the screen to display a label or prompt string defining what this field should contain.
- Another `JTextField` object called `output` which defines a field on the screen in which the greeting can be displayed (gray outlined box in Figure 10.1 or Figure 10.2).

Later we will also use `JButton` objects which are buttons that respond to mouse clicks.

Now we can declare instance variables for these three components as follows

```
private JLabel prompt;
private JTextField input;
private JTextField output;
```

These variables are the names of the GUI component objects.

10.3.3 Creating the GUI components

In the `Greeting1` constructor we can create the objects as follows

```
prompt = new JLabel("Enter your name and press Enter");
input = new JTextField(20);
output = new JTextField(20);
output.setEditable(false);
```

These statements define a `JLabel` object with the given string as label and two `JTextField` objects that can hold about 20 characters each. By default a text field permits input. In our case the `output`

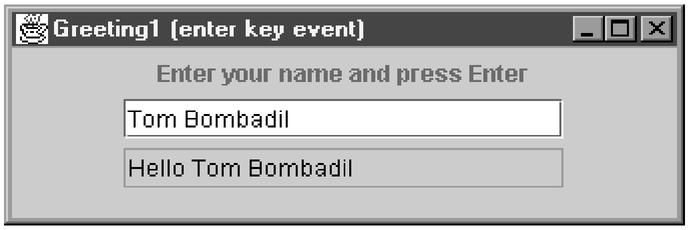


Figure 10.3: Greeting application illustrating the flow layout

field should not accept input so the `setEditable` method is used with the `output` field. A `false` argument indicates that input is not allowed and a `true` argument indicates that input is allowed. You can visually distinguish an output-only text field from an input field.

10.3.4 Choosing a layout manager for the GUI components

The laying out of the GUI components is done by a **layout manager** object. Several types of layout managers are available and they are very convenient since it is not necessary to specify the coordinates and sizes of components or determine how they change if the frame is resized.

The default layout manager for a `JFrame` is called `BorderLayout`. However, we want to use the `FlowLayout` manager. It lays out the components in left to right order in the frame, moving components to the next line when there is no room on the current line for the next component. It's just like putting text on a page using word wrap. When the right margin is reached a new line is begun. Here we are laying out components instead of text.

In the application frames shown in Figure 10.1 and Figure 10.2 there are two objects on the first line (the `prompt` and `input` objects) and one on the second line (the `output` object). If you resize the application window with the mouse then the objects flow (hence the name `FlowLayout` manager) to accommodate the new size. For example, a narrower size produces the layout shown in Figure 10.3 having three rows with one object per row.

10.3.5 Adding GUI components to the frame

Once a layout manager has been chosen it is specified using the `setLayout` method and then the `add` method is used to specify the components to the layout manager. The components are added to the content pane of the frame. In our greeting example we need to use the following statements to add components.

```
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(prompt);
cp.add(input);
cp.add(output);
```

A `JFrame` is composed of several layers called panes. The only pane we need to work with is the content pane: when adding components to a `JFrame` we must add them to the frame's content

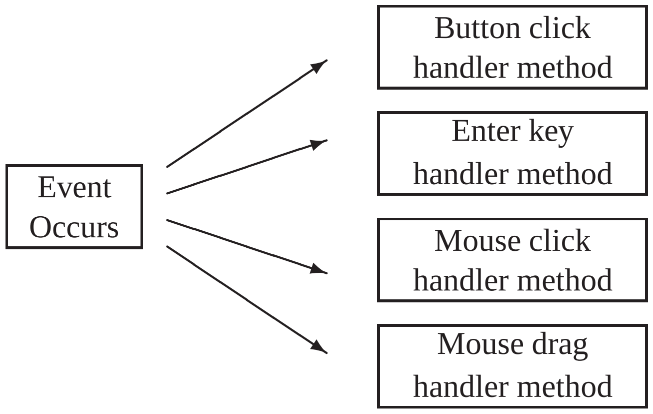

Event Occurs
Button click handler method
Enter key handler method
Mouse click handler method
Mouse drag handler method

Figure 10.4: When an event occurs the component calls the appropriate event handler

pane. The first line gets the frame's content pane which is a `Container` object and declares `cp` to be a reference to it. The next line specifies that a `FlowLayout` manager should be used. The last three lines add the three components to the container. The order is important since it specifies the left to right, top to bottom order that the layout manager will use.

10.3.6 Sending events to listeners

The final step in our GUI design is to indicate what should happen when the Enter key is pressed. This is called **event-driven programming**. In this style of programming we decide what events we want to process, for example, clicking a button with the mouse, pressing Enter in a text field, or dragging the mouse. We must provide some methods that the component can call to process each kind of event. This programming model is quite different from that which we have been doing so far (e.g., get some input, call a method to do some calculations, call a method to display some results).

In event-driven programming we supply methods to handle events but we do not call them ourselves (we have already seen two examples: we have written `main` methods and `paintComponent` methods but we never call them). Instead the component calls them when the event occurs. This is illustrated in Figure 10.4. When an event occurs, represented by the box on the left, the component calls the appropriate event handler method. Four such methods are indicated by the four boxes on the right. The arrows represent the calling of the method.

The information describing each event is stored in an event object. For example, when the user clicks a button or presses the Enter key in a text field, the event is called an action event and the information describing this event is stored in an `ActionEvent` object.

How does the component which receives the event know which event handler object contains the method it should call? The answer is simple. Each event handler implements one of the `EventListener` interfaces. In the case of a button or a text field this is the `ActionListener` interface. For the mouse it could be one or both of the `MouseListener` or `MouseMotionListener` interfaces.

Internally the component maintains a list of references to the event handler objects that want

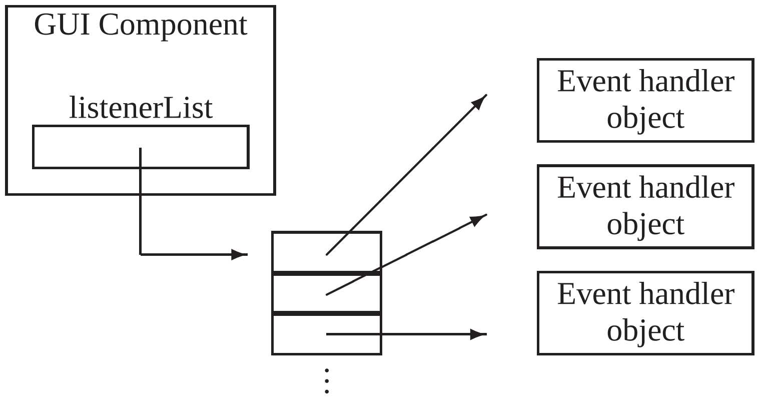


Figure 10.5: Adding listeners (event handlers) to a GUI component

to listen for events generated by the component. This is illustrated in Figure 10.5. When the event occurs the component uses its list of listeners to find the event handling objects that have the appropriate method to call.

In our greeting example the `JTextField` object called `input` needs to know that our application wants to listen for events generated by the pressing of the Enter key. This is done using the statement

```
input.addActionListener(this);
```

The `addActionListener` method simply adds a listener to the list of listeners maintained by the `JTextField` object and in this case the application class itself, referred to by `this`, is the listener: the first line of our `Greeting1` class specifies that the class implements the `ActionListener` interface.

Finally we need to know that any class that implements the `ActionListener` interface must provide an implementation of the `actionPerformed` event handling method having the form

```
public void actionPerformed(ActionEvent e)
{
   // statements to process action events go here
}
```

When the Enter key is pressed after entering text in a `JTextField` object this method is called. The `ActionEvent` object argument describes the details of the event. In particular it can be used to determine which component generated the action event. In our example we have only one component generating action events, so we do not need to use the `ActionEvent` object `e`. Nevertheless, it must be present as an argument since it is part of the interface specification.

10.3.7 Writing event processing code

We are almost finished with our introductory `Greeting1` application. We just need to put some statements into the body of the `actionPerformed` method that will display the greeting in the box associated with the `output` object. This is easy because `JTextField` objects have a `getText` method which returns the text in the field as a `String` object and a `setText` method which takes a `String` object argument and sets the field to that text. These methods have prototypes

```
public String getText();           // return text from a text field
public void setText(String text);   // display given text in a text field
```

Therefore the body of the `actionPerformed` method is

```
String name = input.getText();
output.setText("Hello " + name);
```

which could even be done with the single statement

```
output.setText("Hello " + input.getText());
```

In terms of the message passing terminology we send the `getText` message to the `input` object and the `setText` message to the `output` object so these two text field objects communicate by message passing. Here is the complete application class:

Class `Greeting1`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Using the enter key to trigger an event.
 */
public class Greeting1 extends JFrame implements ActionListener
{
   private JLabel prompt;
   private JTextField input;
   private JTextField output;

   public Greeting1()
   {
      setTitle("Greeting1 (enter key event)");

      prompt = new JLabel("Enter your name and press Enter");
      input = new JTextField(20);
      output = new JTextField(20);
      output.setEditable(false);

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(prompt);
      cp.add(input);
      cp.add(output);

      input.addActionListener(this);
      setSize(450,100);
   }

   /**
    * This method is called when enter key is pressed.
    * @param e the action event that occurred
    */
   public void actionPerformed(ActionEvent e)
   {
      String name = input.getText();
      output.setText("Hello " + name);
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Greeting1();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

It is important to realize that the `input` and `output` variables must be declared as data fields since they are needed in both the constructor and the `actionPerformed` method.

However, the `prompt` declaration could have been moved inside the constructor as a local variable using

```
      JLabel prompt = new JLabel("Enter your name and press Enter");
```

since it is not needed outside the constructor. In fact we could go further and not give it a name at all by replacing

```
      cp.add(prompt);
```

with the statement

```
      cp.add(new JLabel("Enter your name and press Enter"));
```

As a matter of style it is best to declare all GUI variables as data fields, as we have done in the greeting application.

10.3.8 Using a button in the greeting application

As a variation of the `Greeting1` application let us use a button object, instead of the Enter key, to generate the action event that produces the greeting in the `output` field. We will call it the `Greeting2` application. The application frame is shown in Figure 10.6. We can modify `Greeting1` to obtain `Greeting2`. First add a reference to a `JButton` using

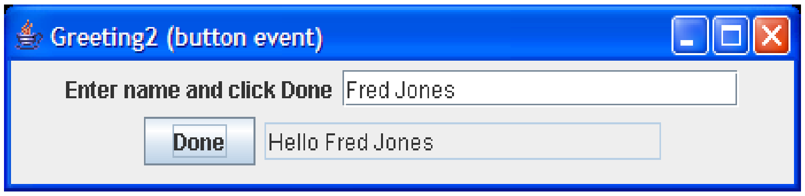


Figure 10.6: Greeting application containing a button

```
    private JButton done;
```

In the constructor define the button using

```
    done = new JButton("Done");
```

which specifies that the button should have the given string displayed on it. Also add it to the frame using

```
    cp.add(done);
```

and finally, replace the statement

```
    input.addActionListener(this);
```

with the statement

```
    done.addActionListener(this);
```

to specify that our application will listen for button click events. Here is the revised application class.

Class `Greeting2`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Using a button to trigger an event.
 */
public class Greeting2 extends JFrame implements ActionListener
{
   private JLabel prompt;
   private JTextField input;
   private JTextField output;
   private JButton done;
   public Greeting2()
   {
      setTitle("Greeting2 (button event)");

      prompt = new JLabel("Enter name and press Enter");
      input = new JTextField(20);
      output = new JTextField(20);
      output.setEditable(false);
      done = new JButton("Done");

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(prompt);
      cp.add(input);
      cp.add(done);
      cp.add(output);

      done.addActionListener(this);
      setSize(450,100);
   }

   /**
    * This method is called when done button is clicked.
    * @param e the action event that occurred
    */
   public void actionPerformed(ActionEvent e)
   {
      String name = input.getText();
      output.setText("Hello " + name);
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Greeting2();
      f.setVisible(true); // make frame visible
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

Now pressing the Enter key has no effect since there are no listeners attached to the `input` field. We could also specify that either pressing the Enter key or the `done` button should cause the action event. This can be done by adding both as listeners (the list in Figure 10.5 has two entries):

```
      input.addActionListener(this);
      done.addActionListener(this);
```

Now the `actionPerformed` method will be called in either case.

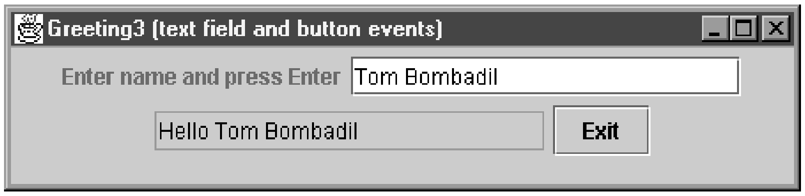

Greeting3 (text field and button events)
Enter name and press Enter
Tom Bombadil
Hello Tom Bombadil
Exit

Figure 10.7: Greeting application containing an exit button

10.3.9 Multiple types of action event responses

As another modification to the `Greeting1` application let us include an `exit` button on the frame. When it is pressed the application should be terminated in the same manner as pressing the close box. As in `Greeting1` the Enter key can be used to signal that the greeting should be displayed in the `output` field. We will call this application `Greeting3`. The application frame is shown in Figure 10.7. Now we run into a problem writing the `actionPerformed` method. This method can be called either when the Enter key is pressed or when the `exit` button is clicked. Since the actions are different in each case we need to know which event occurred. This is where the `ActionEvent` object method argument is useful. We can ask this object which component signaled the event using the `getSource` method in the `ActionEvent` class. The `actionPerformed` method is now given by

```
public void actionPerformed(ActionEvent e)
{
   if (e.getSource() == input) // enter was pressed
   {
      String name = input.getText();
      output.setText("Hello " + name);
   }
   else // exit button must have been clicked
   {
      System.exit(0); // exit program
   }
}
```

Thus, if `getSource` returns the reference `input` then we know the Enter key was pressed. If it returns the reference `exit` we know the `exit` button was pressed.

Another possibility is to use the `getActionCommand` method which returns the string on the button:

```
public void actionPerformed(ActionEvent e)
{
   if (e.getActionCommand().equals("Exit"))
      System.exit(0) // exit program
   else
   {
      String name = input.getText();
```

```
            output.setText("Hello " + name);
         }
      }
```

Here is the complete application class.

Class `Greeting3`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Distinguish different kinds of events
 * using the getSource() method in ActionEvent class.
 */
public class Greeting3 extends JFrame implements ActionListener
{
   private JLabel prompt;
   private JTextField input;
   private JTextField output;
   private JButton exit;

   public Greeting3()
   {
      setTitle("Greeting3 (text field and button events)");

      prompt = new JLabel("Enter name and press Enter");
      input = new JTextField(20);
      output = new JTextField(20);
      output.setEditable(false);
      exit = new JButton("Exit");

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(prompt);
      cp.add(input);
      cp.add(output);
      cp.add(exit);

      input.addActionListener(this);
      exit.addActionListener(this);
      setSize(450,100);
   }

   /**
    * This method is called when the enter key is pressed in the
    * input field or when the exit button is clicked.
    * @param e the action event that occurred
    */
   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource() == input) // enter was pressed
      {
         String name = input.getText();
         output.setText("Hello " + name);
      }
      else // exit button must have been clicked
      {
         System.exit(0); // exit program
      }
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Greeting3();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

10.3.10 Using inner classes to specify event handlers

There is another elegant way to process events that uses inner classes to specify event handlers. An inner class is defined inside another class (called the outer class). An important feature of inner classes is that they can directly access the data fields of the outer class. We will write a variation of `Greeting3` called `Greeting4` that uses inner classes for event handlers. For example, an inner class to handle the pressing of the Enter key in the `input` field is

```
public class EnterKeyHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      String name = input.getText();
      output.setText("Hello " + name);
   }
}
```

This class contains only the `actionPerformed` method and this method refers to the `input` and `output` variables defined in the outer class, namely the `Greeting4` class. We could have made this class an external class in its own file but it would not have access to the `input` and `output` variables – and this is the power of inner classes.

Similarly we write another inner class to handle the closing of the window when the exit button is pressed:

```
public class ExitButtonHandler implements ActionListener
```

```
      {
         public void actionPerformed(ActionEvent e)
         {
            System.exit(0); // exit program
         }
      }
```

We need to add objects of these classes as action listeners so we replace the two statements

```
      input.addActionListener(this);
      exit.addActionListener(this);
```

from `Greeting3` with the statements

```
      input.addActionListener(new EnterKeyHandler());
      exit.addActionListener(new ExitButtonHandler());
```

since the application class (`this`) is no longer handling these events. Therefore it is important to remove the phrase `implements ActionListener` from the `Greeting4` class header and to remove the `actionPerformed` method from the `Greeting4` class since the inner classes are now implementing the `ActionListener` interface. Here is the application class.

Class `Greeting4`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Using inner classes for event handlers
 */
public class Greeting4 extends JFrame
{
   private JLabel prompt;
   private JTextField input;
   private JTextField output;
   private JButton exit;

   public Greeting4()
   {
      setTitle("Greeting4 (text field and button events)");

      prompt = new JLabel("Enter name and press Enter");
      input = new JTextField(20);
      output = new JTextField(20);
      output.setEditable(false);
      exit = new JButton("Exit");
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(prompt);
      cp.add(input);
      cp.add(output);
      cp.add(exit);

      input.addActionListener(new EnterKeyHandler());
      exit.addActionListener(new ExitButtonHandler());
      setSize(450,100);
   }

   /**
    * The actionPerformed method of this inner class will be called
    * whenever the enter key is pressed in the input field.
    */
   public class EnterKeyHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         String name = input.getText();
         output.setText("Hello " + name);
      }
   }

   /**
    * The actionPerformed method of this inner class will be called
    * whenever the exit button is clicked.
    */
   public class ExitButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0); // exit program
      }
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Greeting4();
      f.setVisible(true); // make frame visible
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

Now an if-statement that uses `getSource` is not required to determine the component that triggered the event. The `EnterKeyHandler` object is associated only with the `input` component so its `actionPerformed` method can only be called when the Enter key is pressed.

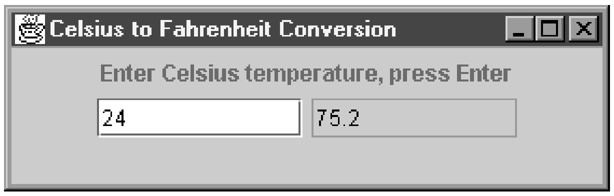

Celsius to Fahrenheit Conversion
Enter Celsius temperature, press Enter
24
75.2

Figure 10.8: The temperature conversion application frame

External names for inner classes

When you compile the `Greeting4` class you will get the following three `class` files:

```
Greeting4.class
Greeting4$EnterKeyHandler.class
Greeting4$ExitButtonHandler.class
```

For inner classes the compiler makes up names consisting of the outer class name, followed by the `$` sign followed by the inner class name.

10.4 Numeric fields and the temperature application

The field of a `JTextField` object always contains a string. The `getText` and `setText` methods are used to get and set the string in the field. For the temperature conversion application the input field will correspond to a temperature in degrees Celsius and the output field will correspond to a temperature in degrees Fahrenheit. When the Enter key is pressed in the input text field, the input temperature is converted to Fahrenheit and displayed in the output text field. The application frame is shown in Figure 10.8 after the Enter key has been pressed. Since the input field contains a string we need to convert this string to a `double` number. Also when the conversion calculation has been performed the Fahrenheit temperature must be converted back to a string so that it can be displayed using `setText` in the output field. These conversions are easily performed.

10.4.1 Numeric and string conversions

Converting numbers to strings

A number can be converted to a string by simply concatenating it with the empty string. For example, if `n` is an `int`, `float` or `double` number, either of the following expressions convert it to a string.

```
"" + n
String.valueOf(n)
```

Therefore if you want to display a number `n` in a `JTextField` called `output` use either of the statements

```
output.setText("" + n);
output.setText(String.valueOf(n));
```

Converting strings to numbers

Conversely, it is easy to convert strings to numbers. For example, to convert a string `s` to an `int` number or a `double` number you can use the expressions

```
Integer.parseInt(s)
Double.parseDouble(s)
```

Therefore if `input` is the name of a `JTextField` object then the string in it can be returned as an `int` value using the statement

```
int i = Integer.parseInt(input.getText().trim());
```

or it can be returned as a `double` value using the statement

```
double d = Double.parseDouble(input.getText().trim());
```

In these statements `trim` is used to remove any leading or trailing spaces that may have been typed. These would give an error in the conversion of the string to a number.

Later we will show how the `JTextField` class can be extended to an `InputJTextField` class that has `getInt` and `getDouble` methods to perform these conversions automatically.

Using numeric fields in the temperature application

For the temperature application we need to get the string that the user types in the input box, convert it to a `double` number, compute the corresponding Fahrenheit temperature and convert it back to a string which can be displayed in the output field. This can be done by placing the following statements in the `actionPerformed` method.

```
double tc = Double.parseDouble(input.getText().trim());
double tf = (9.0/5.0) * tc + 32.0;
output.setText("" + tf);
```

10.4.2 Temperature application

We can now easily modify the `Greeting4` class to produce the temperature application: we need a `JLabel` object called `prompt` to label the input field, a `JTextField` called `input` for the Celsius input temperature, and a `JTextField` called `output` for the converted Fahrenheit temperature. Here is the application class.

Class `Temperature`

`book-project/chapter10/simple`

```
package chapter10.simple;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
/**
 * Converting a Celsius temperature to a Fahrenheit.
 */
public class Temperature extends JFrame
{
   private JLabel prompt;
   private JTextField input;
   private JTextField output;

   public Temperature()
   {
      setTitle("Celsius to Fahrenheit Conversion");
      setSize(325,100);

      prompt = new JLabel("Enter Celsius temperature, press Enter");
      input = new JTextField(10);
      output = new JTextField(10);
      output.setEditable(false);

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(prompt);
      cp.add(input);
      cp.add(output);

      input.addActionListener(new EnterKeyHandler());
   }

   /**
    * The actionPerformed method of this inner class will be called
    * whenever the enter key is pressed in the input field.
    */
   public class EnterKeyHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         double tc = Double.parseDouble(input.getText().trim());
         double tf = (9.0/5.0)*tc + 32.0;
         output.setText("" + tf);
      }
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Temperature();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

10.5 Multi-line text fields

10.5.1 `JTextArea` objects

A `JTextField` object is limited to displaying one line of text. If the text is too long for the field it can be scrolled left or right. To obtain a multi-line field it is necessary to use a `JTextArea` component object which appears as a rectangular area in the window. These objects do not automatically provide for scrolling up and down or left and right in case the text does not fit the rectangular area. Therefore the text area will be added to a `JScrollPane` object to provide the scroll bars if needed.

10.5.2 Investment application

As an illustration let us develop an application for displaying the future value of an investment given an annual rate r in percent, and an initial investment amount a. We will assume that interest is compounded monthly so the future value f of the investment after n years ($12n$ months) is

$$f = a\left(1 + \frac{r}{1200}\right)^{12n}$$

The application will provide input fields for the rate r and the initial amount a and will display the future value for years 1 to 30 in a `JTextArea` object. The Enter key cannot be used to signal that the input has been entered since there are now two input fields. Instead we will use a button to signal that the results should be calculated.

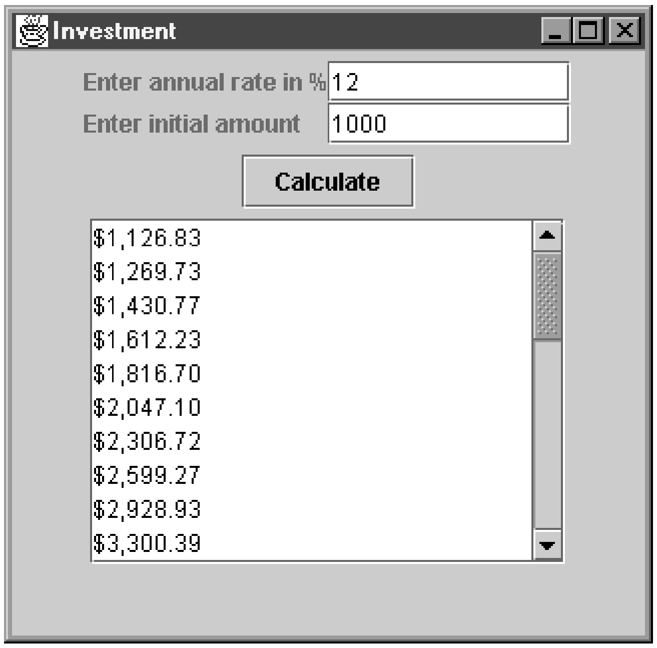
We need 6 component objects: a label called `prompt1` and its associated input field called `rateField` for the annual rate r, another label called `prompt2` and its associated input field called `amountField` for the initial amount a, a button called `calculate` to press when the input data has been entered, and finally an output text area called `output` to display the iterations. The application frame should appear as in Figure 10.9 after the data has been entered and the `calculate` button has been pressed. The following statements declare references to the six GUI objects:

```
private JLabel prompt1;
private JLabel prompt2;
private JTextField rateField;
private JTextField amountField;
private JButton calculate;
private JTextArea output;
```

In the constructor they can be initialized using the statements

```
prompt1 = new JLabel("Enter annual rate in %");
rateField = new JTextField("12", 10);
prompt2 = new JLabel("Enter initial amount");
amountField = new JTextField("1000", 10);
calculate = new JButton("Calculate");
output = new JTextArea(10,20); // 10 rows and 20 columns
output.setEditable(false);
```

Here we have used another variation of the `JTextField` constructor that has two arguments: the first is a string that should appear in the field initially, and the second is the approximate field width in characters. In our example the default initial rate is `12` and the default initial amount is `1000`.



```
$1,126.83
$1,269.73
$1,430.77
$1,612.23
$1,816.70
$2,047.10
$2,306.72
$2,599.27
$2,928.93
$3,300.39
```

Figure 10.9: The investment application

Using panels to design GUI layouts

In Figure 10.9 the two input boxes and their prompts line up in the form of a two by two grid. If you add the components to the application using the statements

```
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(prompt1);
cp.add(rateField);
cp.add(prompt2);
cp.add(amountField);
cp.add(calculate);
cp.add(output);
```

you won't be able to achieve this grid style layout by resizing the frame. Also, as mentioned above, if the text area becomes too small text will be lost since the text area object does not scroll.

To obtain the grid style layout for the first four GUI components it is first necessary to use a `JPanel` object that uses a `GridLayout` manager. The purpose of a `JPanel` object is to organize components. The following statements can be used to obtain the layout shown in Figure 10.9.

```
JPanel p = new JPanel();
p.setLayout(new GridLayout(2,2));
p.add(prompt1);    // add prompt to panel in row 1, column 1
p.add(rateField);  // add rateField to panel in row 1, column 2
p.add(prompt2);    // add prompt2 to panel in row 2, column 1
p.add(amountField); // add amountField to panel in row 2, column 2
```

The first two statements define a `JPanel` object that will lay out its four components using a 2 by 2 grid specified by a `GridLayout` manager. Instead of six components we now have only three: the panel, the calculate button and the output text area.

These three components can be added to the content pane of the frame using the `FlowLayout` manager with the statements

```
Container cp = getContentPane();
cp.setLayout(new FlowLayout());
cp.add(p); // add the panel
cp.add(calculate);
cp.add(new JScrollPane(output));
```

The last statement constructs a `JScrollPane` object for the `output` text area and adds it to the content pane. Now scroll bars will appear in either or both directions if needed.

Doing the calculations

The heart of the investment application is a method called `doIterations` which obtains the `yearlyRate` and `initialAmount` by getting the strings in the two input fields and converting them to `double` numbers. Then a for-loop is used to calculate the future value at the end of each year for 30 years:

```
private void doIterations()
{
   NumberFormat currency = NumberFormat.getCurrencyInstance();
   double yearlyRate =
      Double.parseDouble(rateField.getText().trim());
   double initialAmount =
      Double.parseDouble(amountField.getText().trim());

   output.setText(""); // clear text in area

   double amount = initialAmount;
   for (int year = 1; year <= 30; year++)
   {
      amount = futureValue(amount, yearlyRate, 1);
      output.append(currency.format(amount) + "\n");
   }
}
```

Here we use a `currency` object that knows how to format numbers. In an English locale numbers are formatted with commas separating the thousands, a dollar sign prefix, and rounding to the nearest cent. To make the `NumberFormat` class available it is necessary to use the `import` statement

```
import java.text.NumberFormat;
```

at the top of the class.

There are two methods which can be used to display text in a text area. The `setText` method works the same as for text fields. It replaces all the text in the text area by the given text. It is used in the above method to clear any text from a previous calculation:

```
output.setText("");
```

To display text in the text area we do not use the familiar `print` and `println` console output methods. Instead the `append` method is used. It adds text at the end of any text already displayed in the area. To obtain a new line it is necessary to use `"\n"` in the string. Therefore, in the `doIterations` method the statement

```
output.append(currency.format(amount) + "\n");
```

is used to display the amount and move to the next line.

Finally, an inner class can be used to implement the `ActionListener` interface.

Here is the complete investment application class.

Class `Investment`

book-project/chapter10/investment

```
package chapter10.investment;
import java.text.NumberFormat;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * A gui application for the future value of an investment
 */
public class Investment extends JFrame
{
   private JLabel prompt1;
   private JLabel prompt2;
   private JTextField rateField;
   private JTextField amountField;
   private JButton calculate;
   private JTextArea output;

   public Investment()
   {
      setTitle("Investment");
      setSize(325,320);

      prompt1 = new JLabel("Enter annual rate in %");
      rateField = new JTextField("12", 10);
      prompt2 = new JLabel("Enter initial amount");
      amountField = new JTextField("1000", 10);
      calculate = new JButton("Calculate");
      output = new JTextArea(10,20); // 10 rows and 20 columns
      output.setEditable(false);

      // Make 2 by 2 grid for prompts and inputs

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(2,2));
      p.add(prompt1);
      p.add(rateField);
      p.add(prompt2);
      p.add(amountField);

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(p);
      cp.add(calculate);
      cp.add(new JScrollPane(output));

      calculate.addActionListener(new CalculateButtonHandler());
      doIterations(); // do calculations for default initial values
   }

   private void doIterations()
   {
      NumberFormat currency = NumberFormat.getCurrencyInstance();
      double yearlyRate =
         Double.parseDouble(rateField.getText().trim());
      double initialAmount =
         Double.parseDouble(amountField.getText().trim());

      output.setText(""); // clear text in area

      double amount = initialAmount;
      for (int year = 1; year <= 30; year++)
      {
         amount = futureValue(amount, yearlyRate, 1);
         output.append(currency.format(amount) + "\n");
      }
   }

   private static double futureValue(double amount,
      double yearlyRatePercent, int years)
   {
      double monthlyRate = yearlyRatePercent / 100.0 / 12.0;
      double a =
         amount * Math.pow(1.0 + monthlyRate, 12 * years);
      return a;
   }

   /**
    * The actionPerformed method of this inner class will be called
    * whenever the calculate button is clicked.
    */
   public class CalculateButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         doIterations();
      }
   }

   /** Construct an application and make it visible
    */
   public static void main(String[] args)
   {
      JFrame f = new Investment();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

When the `calculate` button is clicked the system will call the `actionPerformed` method and the calculations will be performed. An interesting feature in this program is the `doIterations` call at the end of the constructor. This means that when the application runs it will do the calculations automatically for the given default values without having to click the button.

10.6 Using inheritance to design smarter text fields

One of the deficiencies of a `JTextField` object is that it only provides a `getText` method to return the text in the field as a string. In many applications, such as `Investment`, the fields are interpreted either as `int` or `double` values. It would be nice to have `getInt` and `getDouble` methods that would automatically convert the data typed in a text field to an `int` value and a `double` value, respectively. Also, if a user types a non-numeric value in a field that is expecting a numeric value then a `NumberFormatException` occurs. A simple way to prevent this is to catch the exception and simply replace the invalid input typed in the field by the number 0 or an error string.

We can easily obtain a smarter text field by using inheritance to extend the `JTextField` class. We will call our subclass `InputJTextField`. To make it more complete we will also provide `getLong` and `getFloat` methods for converting to the `long` and `float` types and we will provide a `getString` method which is like `getText` except that leading and trailing spaces will be removed.

10.6.1 Structure of the `JTextField` class

According to the documentation for the `JTextField` class its basic structure is

```
public class JTextField extends JTextComponent implements SwingConstants
{
   public JTextField() {...}
   public JTextField(String text) {...}
   public JTextField(int columns) {...}
   public JTextField(String text, int columns) {...}
```

```
      // other methods that will be inherited
   }
```

Since constructors are not inherited we can design our subclass to have the same four kinds of constructors. Therefore our subclass will have the structure

```
   public class InputJTextField extends JTextField
   {
      public InputJTextField() {...}
      public InputJTextField(String text) {...}
      public InputJTextField(int columns) {...}
      public InputJTextField(String text, int columns) {...}

      // Here are our new methods

      public int getInt() {...}
      public long getLong() {...}
      public float getFloat() {...}
      public double getDouble() {...}
      public String getString() {...}
   }
```

Since we are not introducing any new data fields the constructors can easily be implemented with the appropriate `super` constructor call expression. To implement `getInt` we can use a `try-catch` block:

```
   public int getInt()
   {
      try
      {
         return Integer.parseInt(getText().trim());
      }
      catch (NumberFormatException e)
      {
         setText("0");
         return 0;
      }
   }
```

The other three get methods are similar and the complete class is

Class `InputJTextField`

`book-project/custom_classes`

```
package custom_classes;
import javax.swing.*;

/**
```

```
 * An InputJTextField is just like a JTextField but it provides methods
 * for reading int, long, float and double numbers in the field:
 * getInt(), getLong(), getFloat(), and getDouble() return the contents
 * of the TextField as in int, long, float, or a double.
 * A clear() method for clearing the text in the field is also provided
 * <p>
 * EXAMPLE
 * <pre>
 *    InputJTextField field = new InputJTextField(20);
 *    ...
 *    int n = field.getInt();       // return field as an int
 *    long l = field.getLong();     // return field as a long
 *    float f = field.getFloat();   // return field as a float
 *    double d = field.getDouble(); // return field as a double
 * </pre>
 * For completeness a method for getting a string is also
 * provided. getString() corresponds to getText() except that leading and
 * trailing blanks are removed.
 */

public class InputJTextField extends JTextField
{
   public InputJTextField()
   {
      super();
   }

   public InputJTextField(String s)
   {
      super(s);
   }

   public InputJTextField(int columns)
   {
      super(columns);
   }

   public InputJTextField(String s, int columns)
   {
      super(s, columns);
   }

   /**
    * Return contents of the field as an int.
    * @return the contents of the field as an int
    */
   public int getInt()
   {
      try
      {
         return Integer.parseInt(getText().trim());
      }
      catch (NumberFormatException e)
      {
         setText("0");
         return 0;
      }
   }

   /**
    * Return contents of the field as a long int.
    * @return the contents of the field as a long int
    */
   public long getLong()
   {
      try
      {
         return Long.parseLong(getText().trim());
      }
      catch (NumberFormatException e)
      {
         setText("0");
         return 0;
      }
   }

   /**
    * Return the contents of the field as a float.
    * @return the contents of the field as a float
    */
   public double getFloat()
   {
      try
      {
         return Float.parseFloat(getText().trim());
      }
      catch (NumberFormatException e)
      {
         setText("0");
         return 0;
      }
   }

   /**
    * Return the contents of the field as a double.
    * @return the contents of the field as a double
    */
   public double getDouble()
   {
      try
      {
         return Double.parseDouble(getText().trim());
      }
      catch (NumberFormatException e)
      {
         setText("0");
         return 0;
      }
   }

   /**
    * Return the contents of the field as a String
    * with leading and trailing blanks are removed.
    * @return the contents of the field as a String
    * with leading and trailing spaces removed.
    */
   public String getString()
   {
      return getText().trim();
   }
}
```

We can use this class to write a new version of the `Investment` application: Simply replace the two `JTextField` objects `rateField` and `amountField` with `InputJTextField` objects. In the `doIterations` method replace the second and third statements with

```
double yearlyRate = rateField.getDouble();
double initialAmount = amountField.getDouble();
```

10.7 GUI for the loan repayment class

In Chapter 7 (page 346) we developed the `LoanRepaymentTable` class that produced a loan repayment table. The table was produced as one big string and returned using the `toString` method. We wrote an application class called `LoanRepaymentTableRunner` (page 348) to run the class either from the console (command prompt or terminal window) or from `BlueJ`.

10.7.1 GUI version of the loan repayment class

Now we want to produce a GUI version called `LoanRepaymentTableGUI`. The important idea is that the `LoanRepaymentTable` class from Chapter 7 can be used unchanged for the GUI version: it's a “plug and play” component.

To make a GUI for this class we need four input fields, a calculate button to trigger the calculations, and a text area to hold the table output. When the calculate button is clicked the `actionPerformed` method is called and all calculations are done using the method

```
public void doCalculations()
{
   double a = loanAmountField.getDouble();
   int y = yearsField.getInt();
   int p = paymentsPerYearField.getInt();
   double r = annualRateField.getDouble();
   LoanRepaymentTable table = new LoanRepaymentTable(a,y,p,r);
```

```
         output.setText(table.toString());
      }
```

where `a` is the loan amount, `y` is the number of years, `p` is the number of payments per year, and `r` is the annual interest rate in percent. These values are obtained from four `InputJTextField` objects. Then if `output` is the name of a `JTextArea` object the loan repayment table's `toString` method can be used to display the table in the text area.

We also need to set the font that displays text in the output area. The default font is not a mono-spaced font so the columns will not line up properly. The statement

```
      output.setFont(new Font("Courier", Font.PLAIN, 11));
```

is used to change the output text area font to Courier, which is a mono-font, using plain style (rather than bold) and using a size of 11 points. Here is the complete GUI class.

Class `LoanRepaymentTableGUI`

`book-project/chapter10/loan_repayment`

```
package chapter10.loan_repayment;
import custom_classes.InputJTextField;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * GUI interface for the LoanRepaymentTable class
 */
public class LoanRepaymentTableGUI extends JFrame
{
   private JLabel loanAmountLabel;
   private InputJTextField loanAmountField;
   private JLabel yearsLabel;
   private InputJTextField yearsField;
   private JLabel paymentsPerYearLabel;
   private InputJTextField paymentsPerYearField;
   private JLabel annualRateLabel;
   private InputJTextField annualRateField;

   private JButton calculate;
   private JTextArea output;

   public LoanRepaymentTableGUI()
   {
      setTitle("Loan Repayment");
      setSize(500,450);

      // Construct the four input text fields and their labels

      loanAmountLabel = new JLabel("Loan amount", JLabel.CENTER);
      loanAmountField = new InputJTextField("10000", 10);

      yearsLabel = new JLabel("Years", JLabel.CENTER);
      yearsField = new InputJTextField("10", 5);

      paymentsPerYearLabel = new JLabel("Payments/year", JLabel.CENTER);
      paymentsPerYearField = new InputJTextField("2", 5);

      annualRateLabel = new JLabel("Annual rate %", JLabel.CENTER);
      annualRateField = new InputJTextField("10", 10);

      // Construct button that causes calculations to be performed

      calculate = new JButton("Calculate");

      // Construct the output text area and choose a mono-spaced font
      // so the columns will line-up properly

      output = new JTextArea(20,60); // 10 rows and 20 columns
      output.setEditable(false);
      output.setFont(new Font("Courier", Font.PLAIN, 11));

      // Add input fields and labels to a panel in a 2 by 4 grid

      JPanel p = new JPanel(new GridLayout(2,4));
      p.add(loanAmountLabel);
      p.add(loanAmountField);
      p.add(yearsLabel);
      p.add(yearsField);
      p.add(paymentsPerYearLabel);
      p.add(paymentsPerYearField);
      p.add(annualRateLabel);
      p.add(annualRateField);

      // Add panel, button, and scrollable text area to frame’s content pane

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(p);
      cp.add(calculate);
      cp.add(new JScrollPane(output));

      // Add the listeners to the button

      calculate.addActionListener(new CalculateButtonHandler());


      // initialize calculations for default set of input values

      doCalculations();
   }
   public void doCalculations()
   {
      double a = loanAmountField.getDouble();
      int y = yearsField.getInt();
      int p = paymentsPerYearField.getInt();
      double r = annualRateField.getDouble();
      LoanRepaymentTable table = new LoanRepaymentTable(a,y,p,r);
      output.setText(table.toString());
   }

   /**
    * A class to implement the actionPerformed method which will be called
    * when the calculate button is pressed.
    */
   public class CalculateButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         doCalculations();
      }
   }

   public static void main(String[] args)
   {
      JFrame f = new LoanRepaymentTableGUI();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

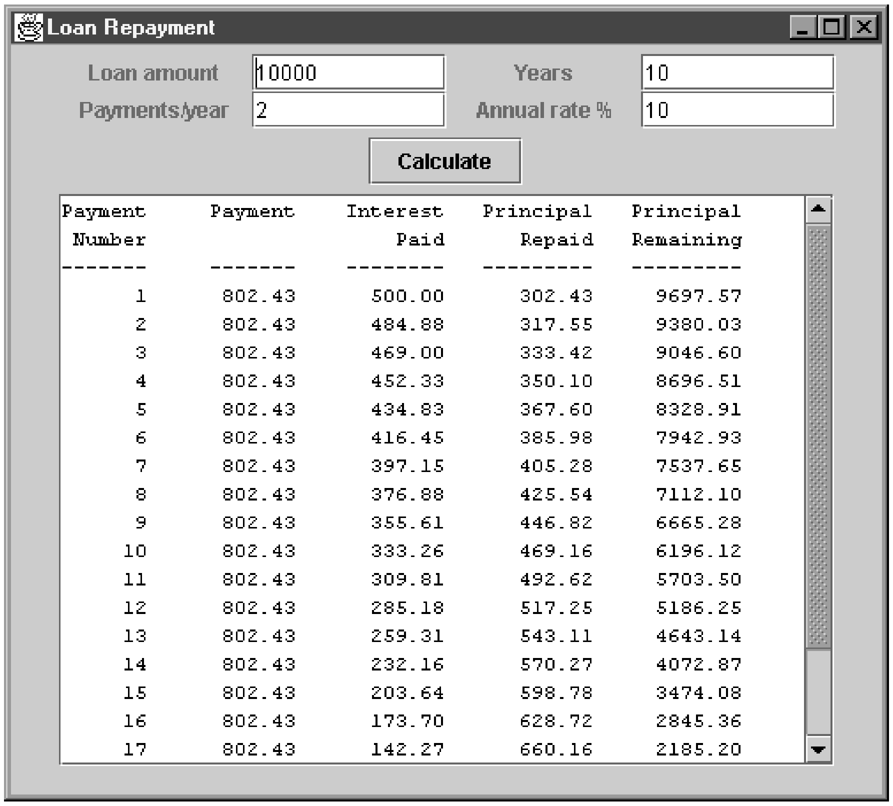
The output window is shown in Figure 10.10.

We have used a two-argument `JLabel` constructor (see documentation) whose second argument is one of these constants indicating that the label should be centered in its field. The default value used by the one-argument constructor is `JLabel.RIGHT`.

10.8 Unit conversion application

As another interesting problem let us write an application called `Conversions` to convert numbers from one unit to another. There will be an input text field for the number to convert and an output text field for the converted result. We can use our custom `InputJTextField` for the input field. Each type of conversion, such as centimeters to inches or miles to kilometers will be represented by a button. We will implement twelve different types of conversions using three rows of four buttons. The initial application frame is shown in Figure 10.11. The frame contains a `JPanel` containing 16 components laid out in a four by four grid using a `GridLayout` manager. The first row is different from the others and the remaining rows can each have 4 buttons. Corresponding to these buttons there will be an array of strings for the button names and an array of `double` numbers for the conversion factors.

Here are the private data fields for our `Conversions` class:



Loan amount: 10000
Years: 10
Payments/year: 2
Annual rate %: 10

```
Payment     Payment    Interest   Principal   Principal
 Number                   Paid      Repaid   Remaining
-------     -------   --------   ---------   ---------
      1      802.43     500.00      302.43     9697.57
      2      802.43     484.88      317.55     9380.03
      3      802.43     469.00      333.42     9046.60
      4      802.43     452.33      350.10     8696.51
      5      802.43     434.83      367.60     8328.91
      6      802.43     416.45      385.98     7942.93
      7      802.43     397.15      405.28     7537.65
      8      802.43     376.88      425.54     7112.10
      9      802.43     355.61      446.82     6665.28
     10      802.43     333.26      469.16     6196.12
     11      802.43     309.81      492.62     5703.50
     12      802.43     285.18      517.25     5186.25
     13      802.43     259.31      543.11     4643.14
     14      802.43     232.16      570.27     4072.87
     15      802.43     203.64      598.78     3474.08
     16      802.43     173.70      628.72     2845.36
     17      802.43     142.27      660.16     2185.20
```

Figure 10.10: `LoanRepaymentTableGUI` application

```
private InputJTextField input;    // source amount
private JTextField output;        // converted amount
private JLabel inputLabel;
private JLabel outputLabel;

private String[] buttonNames = {"in to cm", "cm to in", "ft to cm",
   "cm to ft", "mi to km", "km to mi", "gal to li", "li to gal",
   "lb to gm", "gm to lb", "oz to gm", "gm to oz"};

private double[] factor = {2.54, 1/2.54, 30.5,
   1/30.5, 1.609, 1/1.609, 3.785, 1/3.785,
   453.6, 1/453.6, 28.3495, 1/28.3495};
```

The four fields in the top row are declared first. They provide for the input text field, the output text field and labels for each of them. Next come the arrays for the 12 types of conversions. The first, `buttonNames`, provides an array of button labels indicating the type of conversion. The next gives the conversion factor to use. For example the factor `2.54` is the conversion factor from inches to centimeters.

These components are constructed and initialized inside the `Conversions` constructor. The

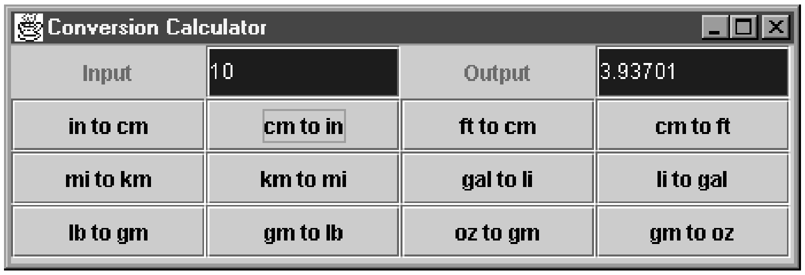


Figure 10.11: `Conversions` application

labels and text fields are constructed with the statements

```
inputLabel = new JLabel("Input", JLabel.CENTER);
input = new InputJTextField("1", 10);
outputLabel = new JLabel("Output", JLabel.CENTER);
output = new JTextField(10);
```

Next we need to determine how many rows of buttons there are (3 rows for 12 types of conversions). To make it easy to modify the application to include more or less than 12 types of conversions the number of rows can be computed using

```
int rows = buttonNames.length / 4;
if (buttonNames.length % 4 != 0)
   rows++;
```

Next we can define a `JPanel` with `GridLayout` manager and lay out the first row of components:

```
JPanel p = new JPanel();
p.setLayout(new GridLayout(rows + 1, 4));
p.add(inputLabel);
p.add(input);
p.add(outputLabel);
p.add(output);
```

Finally we can use a loop to construct the buttons, use a custom `JButtonHandler` inner class to associate button `i` with an `actionPerformed` method, add the buttons to the panel, and then add the panel to the content pane of the frame:

```
for (int i = 0; i < buttonNames.length; i++)
{
   JButton b = new JButton(buttonNames[i]);
   b.addActionListener(new JButtonHandler(i));
   p.add(b);
}
Container contentPane = getContentPane();
contentPane.add(p);
```

Notice that the `JButtonHandler` class has a constructor with one argument, `i`, which is the index of the button. It is not necessary to have an array of buttons and button handlers since the panel will keep track of the buttons and to assign a handler to button `i` we use `new JButtonHandler(i)`. The inner class is given by

```
public class JButtonHandler implements ActionListener
{
   private int buttonIndex;

   public JButtonHandler(int index)
   {
      buttonIndex = index;
   }

   public void actionPerformed(ActionEvent e)
   {
      double in = input.getDouble();
      double out = in * factor[buttonIndex];
      output.setText(String.format("%.5f", out));
   }
}
```

Thus, the index encapsulated by an object of this class can be used as an index into the array of conversion factors in the `actionPerformed` method so the conversion from input units to output units is given by

```
double out = in * factor[buttonIndex];
```

If we had not included a constructor having the button index as argument then we would need an array of buttons, one handler for all buttons, and a for-loop in its `actionPerformed` method that uses `getSource` to determine which button was clicked.

10.8.1 `Conversions` class

Here is the complete application class:

Class `Conversions`

`book-project/chapter10/conversions`

```
package chapter10.conversions;
import custom_classes.InputJTextField;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Conversion from one system of units to another using buttons
 */
public class Conversions extends JFrame
{
   private InputJTextField input;    // source amount
   private JTextField output;        // converted amount
   private JLabel inputLabel;
   private JLabel outputLabel;

   private String[] buttonNames = {"in to cm", "cm to in", "ft to cm",
      "cm to ft", "mi to km", "km to mi", "gal to li", "li to gal",
      "lb to gm", "gm to lb", "oz to gm", "gm to oz"};

   private double[] factor = {2.54, 1/2.54, 30.5,
      1/30.5, 1.609, 1/1.609, 3.785, 1/3.785,
      453.6, 1/453.6, 28.3495, 1/28.3495};

   public Conversions()
   {
      setTitle("Conversion Calculator");

      // instead of setSize we could use f.pack()
      // in the main method

      // setSize(450,150);

      inputLabel = new JLabel("Input", JLabel.CENTER);
      input = new InputJTextField("1", 10);

      outputLabel = new JLabel("Output", JLabel.CENTER);
      output = new JTextField(10);
      output.setEditable(false);

      input.setBackground(Color.blue);
      input.setForeground(Color.white);
      output.setBackground(Color.blue);
      output.setForeground(Color.white);

      // determine number of rows of buttons

      int rows = buttonNames.length / 4;
      if (buttonNames.length % 4 != 0)
         rows++;

      // Construct a panel with a grid layout for the top input/output
      // row and the rows of buttons.

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(rows + 1, 4));
      p.add(inputLabel);
      p.add(input);
      p.add(outputLabel);
      p.add(output);
      // Construct buttons, add listeners to them, and add them to the panel
      // Each listener constructor has an argument defining the button number.

      for (int i = 0; i < buttonNames.length; i++)
      {
         JButton b = new JButton(buttonNames[i]);
         b.addActionListener(new JButtonHandler(i));
         p.add(b);
      }

      // Finally, add the panel to the frame’s content pane

      Container cp = getContentPane();
      cp.add(p);
   }

   /**
    * Handler for button i, i=0,1,2,... Each instance encapsulates a
    * button index which is an index into the conversion factor array.
    */
   public class JButtonHandler implements ActionListener
   {
      private int buttonIndex;

      public JButtonHandler(int index)
      {
         buttonIndex = index;
      }

      public void actionPerformed(ActionEvent e)
      {
         double in = input.getDouble();
         double out = in * factor[buttonIndex];
         output.setText(String.format("%.5f", out));
      }
   }

   public static void main(String[] args)
   {
      JFrame f = new Conversions();
      f.pack();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

10.9 Inheritance and listener interfaces

In Chapter 10 we discussed inheritance, polymorphism and interfaces. We now give further examples using the GUI classes.

10.9.1 `ActionListener` interface

We have seen that an event handler for processing button clicks or the pressing of the Enter key in a text field is an object from a class that implements the `ActionListener` interface. This means that such an event handler "is a type of" `ActionListener`. To add an event handler to a component, such as a button or text field, we use the component's `addActionListener` method which has the prototype

```
public void addActionListener(ActionListener obj)
```

The argument here is polymorphic in the sense that `obj` can be an object from any class that implements the `ActionListener` interface. This interface is defined in the `java.awt.event` package (its full name is `java.awt.event.ActionListener`) as

```
public interface ActionListener extends EventListener
{
   public void actionPerformed(ActionEvent e);
}
```

Therefore a class that implements the `ActionListener` interface only needs to implement the `actionPerformed` method.

This example also shows that interfaces, like classes, can be extended: `ActionListener` extends the `EventListener` interface. If you look at documentation for the `EventListener` superinterface you will see that it has the form

```
public interface EventListener
{
}
```

There are no methods to implement here: the `EventListener` interface simply serves as the superinterface for all listener interfaces such as `ActionListener` and `WindowListener` and many others as well. It is called a "tagging interface".

10.9.2 `WindowListener` interface

We have closed our GUI applications using the statement

```
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

The `WindowListener` interface specifies a more general way to specify what happens when a window is closed, opened, minimizes and restored.

This interface defines methods which are called when a window, such as a `JFrame`, is opened or closed using the minimize, maximize, and close buttons.

The `WindowListener` interface in package `java.awt.event` is more complicated than the `ActionListener` interface because it declares seven methods instead of one. It is defined by

```
public interface WindowListener extends EventListener
{
   public void windowClosing(WindowEvent e);
   public void windowActivated(WindowEvent e);
   public void windowClosed(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowOpened(WindowEvent e);
}
```

Any class that implements this interface must implement all seven methods. This can be inconvenient since normally we don't need all seven methods. Normally we are only interested in the `windowClosing` method.

As a convenience the `WindowAdapter` class in package `java.awt.event` is available as an adapter class. It implements the interface by simply providing empty bodies for the seven methods:

```
public class WindowAdapter implements WindowListener
{
   public void windowClosing(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

Thus we can implement the `WindowListener` interface with a custom `WindowCloser` class obtained by simply by extending the `WindowAdapter` class in package `java.awt.event` and overriding just the `windowClosing` method:

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class WindowCloser extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
      System.exit(0);
   }
}
```

We could have used this class in our GUI programs by including the statement

```
addWindowListener(new WindowCloser());
```

The prototype for the `addWindowListener` method of the `Window` class, a superclass of `JFrame`, is

```
public void addWindowListener(WindowListener obj)
```

10.10 Average mark calculator

We now write a GUI application to calculate the average mark for a student and illustrate how event-driven programming can be used to avoid nested loops typical of ordinary programs that repeatedly process data.

We assume that each student has marks for several tests. We want to enter the marks and compute the average and then process another set of marks for another student.

10.10.1 Console version of average mark calculator

If we write a console application we need to use nested loops as the following program shows:

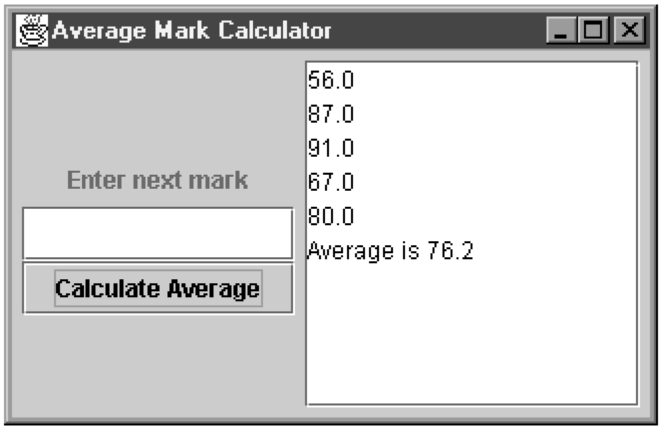
Class `MarkAverageConsole`

`book-project/chapter10/mark_average`

```
package chapter10.mark_average;
import java.util.Scanner;

/**
 * A console application for calculating mark averages.
 * A series of marks for a student are entered and terminated by a
 * negative sentinel mark. Then the user is asked if a set of marks
 * for another student is to be entered. No loops are needed in GUI version.
 */
public class MarkAverageConsole
{
   public MarkAverageConsole()
   {
      Scanner input = new Scanner(System.in);

      // Outer loop processes one of more students

      boolean moreStudents = true;
      while (moreStudents)
      {
         // Inner loop processes marks for a student

         double sumMarks = 0;
         int numMarks = 0;
         System.out.println("Enter marks for a student (neg mark to quit)");
         double mark = input.nextDouble();
         input.nextLine();

         while (mark >= 0)
         {
            sumMarks = sumMarks + mark;
            numMarks = numMarks + 1;
            mark = input.nextDouble();
            input.nextLine();
         }
         double avg = sumMarks / numMarks;
         System.out.println(String.format("Average is %.1f", avg));

         System.out.println("More students Y/N ?");
         char reply = input.nextLine().toUpperCase().charAt(0);
         if (reply == 'N') moreStudents = false;
      }
   }

   public static void main(String[] args)
   {
      new MarkAverageConsole();
   }
}
```

10.10.2 GUI version of average mark calculator

For the GUI version of this program no loops are required. The event-driven programming model takes care of this automatically. We can use a button instead of a sentinel value. Clicking this button is the signal that the marks for a student have been entered and the average should be computed. Then if an average for another student is desired a new set of marks can be entered. Figure 10.12 shows what the GUI application will look like after one set of marks has been entered and Figure 10.13 shows the situation after another set of marks has been entered.



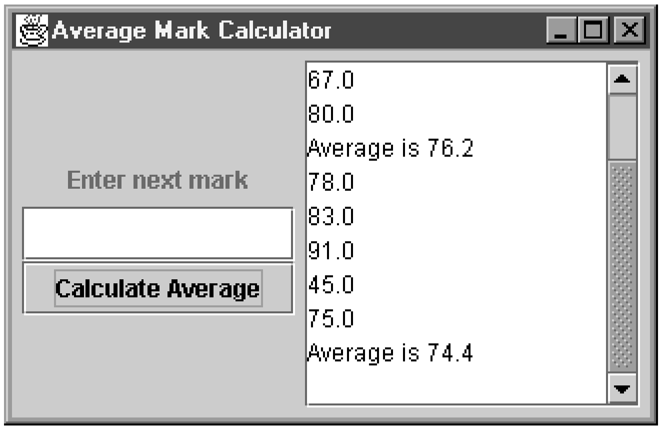
```
56.0
87.0
91.0
67.0
80.0
Average is 76.2
```

Figure 10.12: `MarkAverage` window for one set of marks

Each time a mark is entered in the input text field it is copied to the output text area. When the button is clicked the average is calculated and displayed. Then another set of marks can be entered. As shown in Figure 10.13, the output area automatically scrolls if necessary.

The application contains four components: a label, a field for entering a mark, a button, and a text area. These components can be declared using

```
private JLabel prompt;
```



```
67.0
80.0
Average is 76.2
78.0
83.0
91.0
45.0
75.0
Average is 74.4
```

Figure 10.13: `MarkAverage` window for two sets of marks

```
    private InputJTextField markField;
    private JButton calculate;
    private JTextArea output;
```

and created in the constructor using

```
    prompt = new JLabel("Enter next mark", JLabel.CENTER);
    markField = new InputJTextField("", 10);
    calculate = new JButton("Calculate Average");
    output = new JTextArea(10,15); // 10 rows and 10 columns
    output.setEditable(false);
```

A 3 row and 1 column grid layout panel can be used to organize the label, text field and button in a vertical line. This panel and the text area can be added to the content pane using a flow layout manager:

```
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(3,1));
    p.add(prompt);
    p.add(markField);
    p.add(calculate);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(p);
    cp.add(new JScrollPane(output));
    pack();
```

We have added the output text field to a `JScrollPane` object so that vertical scroll bars will appear if necessary. We have not used `pack` before. There are two ways to specify the size of the frame that holds our components. We have been using `setSize` to do this in our previous GUI applications. Another way is to use `pack` instead of `setSize`. This causes the window to be sized to fit the preferred size and layout of the components. We have not used `pack` previously since its effect with the flow layout manager would be to try to put all components on one line and this would not have looked good. However for the `MarkAverageGUI` application it has the effect of putting the three by one panel and the text area side by side in a frame that just fits and this is what we want. If desired the window can also be resized to move the text area below the button.

The next step is to add action listeners for the button click and for the pressing of the Enter key in the text field. We can use the following inner classes to do this:

```
public class NextMarkFieldHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      addMark();
   }
}

public class CalculateButtonHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      calculateAverage();
   }
}
```

The `addMark` method will be called when the Enter key is pressed. It simply adds the mark to the running sum, increments a counter for the number of marks, copies the mark to the output text area, and clears the input text field in preparation for entering the next mark:

```
private void addMark()
{
   double mark = markField.getDouble();
   sumMarks = sumMarks + mark;
   numMarks++;
   markField.setText("");
   output.append(mark + "\n");
}
```

Finally, the `calculateAverage` method computes the average mark for a student and displays the result in the output text field rounded to two decimal places:

```
private void calculateAverage()
{
   double avg = sumMarks / numMarks;
   output.append(String.format("Average is %.1f\n", avg));
   initialize();
}
```

It also calls an `initialize` method defined by

```
   private void initialize()
   {
      numMarks = 0;
      sumMarks = 0.0;
      markField.setText("");
   }
```

that prepares for the computation of an average for another student. This method can also be used in the constructor to initialize the GUI. Here is the complete application class:

Class `MarkAverageGUI`

`book-project/chapter10/mark_average`

```
package chapter10.mark_average;
import custom_classes.InputJTextField;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * GUI version of MarkAverageConsole.
 */
public class MarkAverageGUI extends JFrame
{
   private JLabel prompt;
   private InputJTextField markField;
   private JButton calculate;
   private JTextArea output;

   private double sumMarks = 0.0;
   private int numMarks = 0;

   public MarkAverageGUI()
   {
      setTitle("Average Mark Calculator");

      prompt = new JLabel("Enter next mark", JLabel.CENTER);
      markField = new InputJTextField("", 10);
      calculate = new JButton("Calculate Average");
      output = new JTextArea(10,15); // 10 rows and 10 columns
      output.setEditable(false);

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(3,1));
      p.add(prompt);
      p.add(markField);
      p.add(calculate);

      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(p);
      cp.add(new JScrollPane(output));

      /* Pack causes the window to be sized to fit the preferred size and
         layouts of its components.
      */
      pack();

      markField.addActionListener(new NextMarkFieldHandler());
      calculate.addActionListener(new CalculateButtonHandler());
      initialize();
   }

   // Prepare for calculation of average for a new student

   private void initialize()
   {
      numMarks = 0;
      sumMarks = 0.0;
      markField.setText("");
   }

   private void calculateAverage()
   {
      double avg = sumMarks / numMarks;
      output.append(String.format("Average is %.1f\n", avg));
      initialize();
   }

   private void addMark()
   {
      double mark = markField.getDouble();
      sumMarks = sumMarks + mark;
      numMarks++;
      markField.setText("");
      output.append(mark + "\n");
   }

   public class NextMarkFieldHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         addMark();
      }
   }

   public class CalculateButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         calculateAverage();
      }
   }

   public static void main(String[] args)
   {
      JFrame f = new MarkAverageGUI();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

10.11 GUI Version of the `RandomTriangles` class

In Chapter 9 we wrote a `RandomTriangles` class (page 497) using a command line argument to get the number of random triangles to draw. If no command line argument was supplied then 10 triangles were drawn. Since `RandomTriangles` is a `JPanel` then, rather than use command line arguments and putting this panel in a `GraphicsFrame`, we can use a text field to input the number of triangles to draw and use a button to signal that the window should be refreshed to use the number in this text field. Thus, the GUI components are the graphics panel, a label, a text field and a button.

10.11.1 `ControlPanel` class

We can think of the last three components as forming a control panel which we can place along the bottom of the frame as shown in Figure 10.14. Above it we can place the `RandomTriangles` graphics panel. The control panel will be an object from a `ControlPanel` class that extends

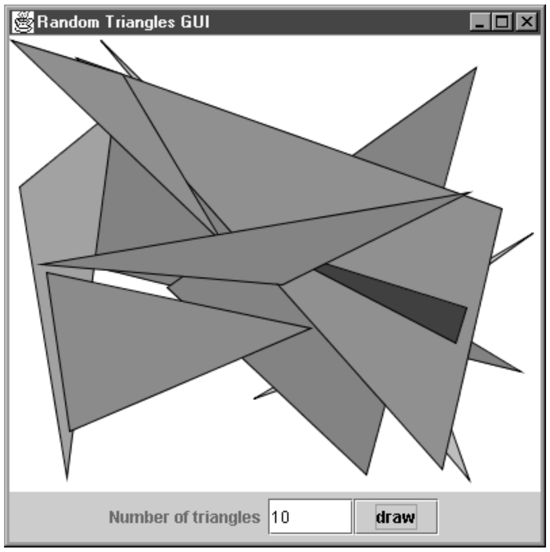


Figure 10.14: The random triangles GUI using a `ControlPanel` and a `JPanel`

`JPanel`. It contains a `JLabel`, an `InputJTextField`, and a `JButton` object, and will communicate with the application class using a `getValue` method that returns the value typed in the text field. The class is given by

Class `ControlPanel`

book-project/chapter10/random_triangles

```
package chapter10.random_triangles;
import custom_classes.InputJTextField;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * A control panel containing a prompt, input text field and a button.
 */
public class ControlPanel extends JPanel
{
   private JButton button;
   private JLabel prompt;
   private InputJTextField inputField;

   public ControlPanel(String promptString, String buttonLabel, int value)
   {
      button = new JButton(buttonLabel);
      prompt = new JLabel(promptString);
      inputField = new InputJTextField("" + value, 5);

      // Use grid layout to make text field and button the same size

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(1,2));
      p.add(inputField);
      p.add(button);

      // put prompt and grid into flow layout (centered is default)

      setLayout(new FlowLayout());
      add(prompt);
      add(p);
   }

   public void addActionListener(ActionListener a)
   {
      button.addActionListener(a);
   }

   public int getValue()
   {
      return inputField.getInt();
   }
}
```

Panels don't have action listeners. They are just used to organize the layout of components and for drawing graphics. However we need an `addActionListener` method here because our panel should respond when the button is clicked. It is easy to write this method: we can have it just add the listener to the button.

We have also made `ControlPanel` more general than it needs to be in this application by using general arguments in the constructor. This class could be reused in any situation where there is a labeled text field and a button to control it.

Now we need to write the `RandomTrianglesGUI` application class. It contains an object called `trianglePanel` from the `RandomTriangles` class and a `ControlPanel` called `controlPanel`. It can implement the `ActionListener` interface and repaint the graphics panel when the draw button is clicked. The `RandomTriangles` class from Chapter 9 can be used unchanged.

This is an excellent example showing how two interacting objects: the `controlPanel` object has a `getValue` method to return the value typed in the text field and the `trianglePanel` object has a `setNumTriangles` method to set the number of triangles that should be drawn. Thus, they can communicate using the statement

```
trianglePanel.setNumTriangles(controlPanel.getValue());
```

This gives the following inner class with an `actionPerformed` method which will be called whenever the control panel's button is clicked.

```
public class ControlPanelHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      trianglePanel.setNumTriangles(controlPanel.getValue());
      trianglePanel.repaint();
   }
}
```

The `repaint` method will cause the graphics panel's `paintComponent` method to be called and this will draw a new set of triangles.

To arrange that the graphics panel fills the entire frame above the control panel we will use a `BorderLayout` for the `RandomTrianglesGUI` class. A `BorderLayout` manager uses five areas of the screen (north, south, east, west, and center) to lay out components. It is not necessary to use all five regions. In any case the center region always expands to use all space not required by the border regions. Therefore the control panel can be added in the south region and the graphics panel can be added to the center region. Here is the complete class.

Class `RandomTrianglesGUI`

book-project/chapter10/random_triangles

```
package chapter10.random_triangles;
```

```
import chapter9.shapes.RandomTriangles;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Show how to place a graphics panel in a frame and have
 * it controlled using a control panel rather than console input.
 */
public class RandomTrianglesGUI extends JFrame
{
   private ControlPanel controlPanel;
   private RandomTriangles trianglePanel;

   public RandomTrianglesGUI()
   {
      setTitle("Random Triangles GUI");
      controlPanel = new ControlPanel("Number of triangles", "draw", 10);
      trianglePanel = new RandomTriangles(10);

      Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      cp.add(trianglePanel, BorderLayout.CENTER);
      cp.add(controlPanel, BorderLayout.SOUTH);

      controlPanel.addActionListener(new ControlPanelHandler());
      setSize(400,400);
   }


   /**
    * The actionPerformed method of this inner class will be called
    * whenever the control panel’s button is clicked.
    */
   public class ControlPanelHandler implements ActionListener
   {
      // When the control panel’s button is clicked, get the value typed
      // in the text field and give it to the RandomTriangles object using
      // its setNumTriangles method.

      public void actionPerformed(ActionEvent e)
      {
         trianglePanel.setNumTriangles(controlPanel.getValue());
         trianglePanel.repaint();
      }
   }

   public static void main(String[] args)
   {
      JFrame f = new RandomTrianglesGUI();
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

This application class shows why we needed to include an addActionListener method in the ControlPanel class. We could have designed all the components into this class and not used a separate ControlPanel class, but using this class shows how to separate the user interaction part of the application from the output part (graphics display).

Many of the graphics programs we have written since Chapter 5 could now be rewritten, using RandomTrianglesGUI as a model, to get their input from the GUI rather than the command line or a Scanner object.

10.12 Inheritance and the GraphicsFrame class

We have used the GraphicsFrame class many times since it made simple graphics programs easy to write. It is another good example of inheritance. Recall from Chapter 5 that classes using GraphicsFrame have the overall structure

```
public class MyClass extends JPanel
{
   // data fields go here

   public void main(String[] args)
   {
      new GraphicsFrame("Title goes here", new MyClass(), 401, 301);
   }

   // other methods go here
}
```

Here the GraphicsFrame constructor arguments are a title string, an instance of the graphics panel class MyClass, and its width and height in pixels. Now a MyClass object “is a” JPanel object and a JPanel object “is a” JComponent object so the GraphicsFrame class has the specification

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class GraphicsFrame extends JFrame
{
   public GraphicsFrame(String title, JComponent c, int w, int h) {...}

   // other methods if needed go here
}
```

The constructor implementation is

```
public GraphicsFrame(String title, JComponent c, int w, int h)
```

```
     {
        super();
        setTitle(title);
        c.setPreferredSize(new Dimension(w,h));
        getContentPane().add(c);
        pack();
        center();
        setVisible(true);
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        }
```

There are a few new features here. First the `setPreferredSize` method is used on the `JComponent` called `c` to set the preferred size of the component using `w` and `h`. This method requires a `Dimension` object as argument. The `Dimension` class in `java.awt` is just a simple class that lets you specify the width and height of components using a single object variable rather than two integer values.

It is also convenient for returning a pair of width and height values as the value of a method (see `getSize` in the `center` method below). The data fields `width` and `height` are public variables so if `d` is a `Dimension` object then `d.width` is the width and `d.height` is the height.

Next `c` is added to the center of the content pane of the frame. Recall that the default layout manager for a frame is `BorderLayout` not `FlowLayout`. This layout manager identifies north, south, east, west, and center positions of a frame or other component, so we specify that the component should go in the center of the frame. Since there are no other components for north, south, east, or west, the component fills all the space in the frame.

Next it is necessary to use `pack` to indicate that the frame should be chosen just large enough to contain the added components (only `c` in our case).

Finally, since frames normally appear in the top left corner of the screen and we want them to appear in the center of the screen, a local `center` method is called to do this.

10.12.1 `GraphicsFrame` class

Here is the complete class.

Class `GraphicsFrame`

`book-project/custom_classes`

```
package custom_classes;
import java.awt.*;
import javax.swing.*;

/**
 * A simple class to set up a graphics frame on which to draw graphics.
 */
 public class GraphicsFrame extends JFrame
{
   /**
```

```
    * Construct a graphics frame with drawing surface of width w pixels and
    * height h pixels.
    * @param title title appearing in the window title bar.
    * @param c the component on which to draw (normally a JPanel object).
    * @param w width of drawing surface in pixels.
    * @param h height of drawing surface in pixels.
    */
   public GraphicsFrame(String title, JComponent c, int w, int h)
   {
      super();
      setTitle(title);
      c.setPreferredSize(new Dimension(w,h));
      getContentPane().add(c);
      pack();
      center();
      setVisible(true);
      setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
   }

   private void center()
   {
      Dimension screenSize = getToolkit().getScreenSize();
      Dimension frameSize = getSize();
      int xLocation = (screenSize.width - frameSize.width) / 2;
      int yLocation = (screenSize.height - frameSize.height) / 2;
      setLocation(xLocation, yLocation);
   }
}
```

The center method uses some fancy tricks to get the screen size and frame size as `Dimension` objects using the `getToolkit` method and then `setLocation` to position the top left corner of the frame so that the frame is centered in the screen.

10.13 Applets

Java application classes are divided into two categories. The first consists of the console and GUI applications that we have written so far. The second consists of GUI applets. It is also possible to write a class that functions both as a GUI application and as an applet.

Applets are special GUI classes that are executed by a Web browser instead of the `java` interpreter. They are compiled in the same way as other Java classes using the `javac` command or using BlueJ. Each applet is specified on a web page using a special HTML applet tag which specifies the class file to run and the width and height in pixels of the applet. This area corresponds to the `JPanel` that we have used in our graphical programs. When the browser finds this tag it allocates the required rectangular area on the screen, loads the specified applet class file (byte codes) and runs the applet. The applet can also be run directly from Bluej.

A template for the simple applets we shall discuss is given by

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
public class MyApplet extends JApplet
{
   // declare variables and GUI components

   public void init()
   {
       // code to execute when the applet is initialized
   }

   public void start() {...}
   public void stop() {...}
   public void destroy() {...}

   // other methods can go here
}
```

This is not the most general form of an applet. There are no constructors in an applet. Instead we override the `init` method of `JApplet`. When the browser runs an applet it executes the `init` method to initialize and display the applet. Therefore any code that normally goes in a constructor is now placed in the `init` method.

There are other methods that are essential for some applets. The browser calls the `start` method each time the user loads the page containing the applet, and it calls the `stop` method when the browser loads a new page. The `destroy` method is called by the browser to perform any clean-up tasks when an applet is being terminated. Many other applet-specific methods are available. For the simplest applets only the `init` method is required.

10.13.1 RGB color applet

As an example, let us write an applet called `RGBColorApplet` that lets us type in three RGB color codes in the range 0 to 255 and display the corresponding color. We will write our class so that it functions both as a GUI application and an applet. We can test applets without using a web browser. A special `appletviewer` program is provided as part of the Java 2 SDK. This program is like a “no-frills” browser whose sole purpose is to test applets. It can be run from the command line. This requires that we write a short HTML file that has an applet tag containing the name of the class file, `RGBColorApplet.class` in our case, and the width and height in pixels of the rectangular panel for the applet and optionally three parameters to describe the initial RGB color values.

For example if you write the following simple HTML file called `TestRGBColorApplet.html` that contains

```
<html>
<body>
<applet code = "RGBColorApplet.class" width="400" height="150">
<param name = "redValue"  value = "125">
```

```
<param name = "greenValue"  value = "205">
<param name = "blueValue"  value = "125">
</applet>
</body>
</html>
```

then the command

```
appletviewer TestRGBColorApplet.html
```

tells `appletviewer` to look in the HTML file for the applet tag giving the class to load and the size of the applet. The applet tag can also be used to parameterize an applet by specifying the initial values of some variables using name-value pairs. Here we use three `PARAM` tags. Each specifies a string for the name of the parameter and a value for this parameter. The applet can read these values using its `getParameter` method. This is analogous to the command line arguments for console applications.

Unlike the Java interpreter, the `appletviewer` program and the web browser completely ignore any classpath you have set since it is doubtful that someone half way around the world will be using your classpath. Only the directory containing the applet class file is searched for custom packages, so in our case we put all required files in the directory that contains `RGBColorApplet.class`.

The `RGBColorApplet` window is shown in Figure 10.15. We will write the applet so that it can

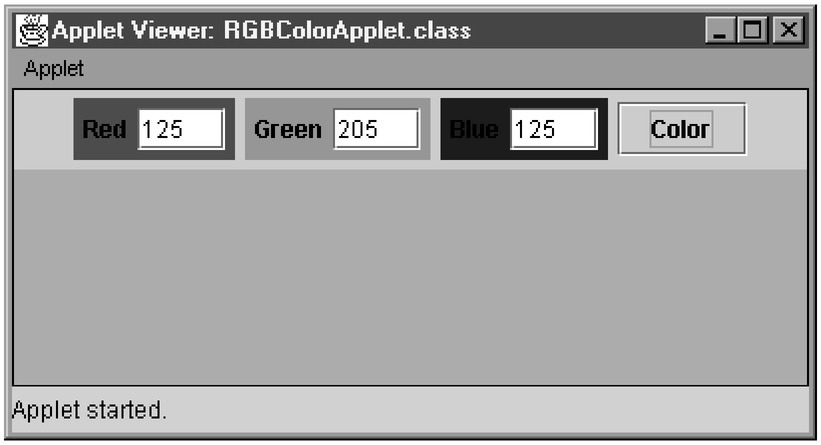


Figure 10.15: `RGBColorApplet` in the applet viewer window

also be run as GUI application using the standard Java interpreter. It is very easy to do this for any applet. Then the command to run the application is

```
java RGBColorApplet
```

and the output window is shown in Figure 10.16.

Laying out the components

The figures show that the frame contains two top-level panels. The top panel is called the control panel, and the large bottom panel is called the color panel. These two panels are declared using

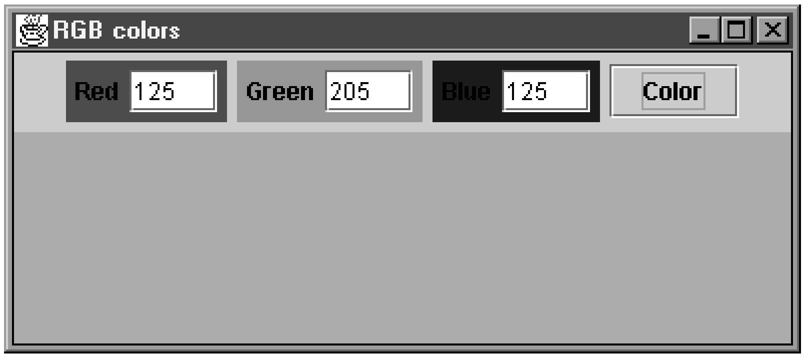


Figure 10.16: `RGBColorApplet` as a GUI application

```
    private JPanel controlPanel;
    private JPanel colorPanel;
```

The control panel also contains three panels, each containing a color label and an input text field, and a color button.

The control panel's three labels, three text fields, and color button are declared using

```
    private JLabel redLabel;
    private JLabel greenLabel;
    private JLabel blueLabel;
    private InputJTextField redField;
    private InputJTextField greenField;
    private InputJTextField blueField;
    private JButton colorButton;
```

The color panel will show the selected color as its background color when the button is clicked.

To get the red, green and blue background rectangles for the labels and text fields it is necessary to put each label and text field in its own panel and set the background color of the panel. Therefore in the `init` method we can construct the control panel and its components. For example, the following statements define the red label, text field, and its red panel.

```
    redLabel = new JLabel("Red", JLabel.CENTER);
    redLabel.setForeground(Color.black);
    redField = new InputJTextField(redValue, 4);
    JPanel pRed = new JPanel();
    pRed.add(redLabel);
    pRed.add(redField);
    pRed.setBackground(Color.red);
```

The variable `redValue` will be a string containing the initial red value. It is obtained from the `PARAM` tag specified in the applet tag. Similarly we can define panels `pGreen` and `pBlue`. The button is constructed using

```
    colorButton = new JButton("Color");
```

The control panel uses the `FlowLayout` manager and can be constructed with the statements

```
controlPanel = new JPanel();
controlPanel.setLayout(new FlowLayout());
controlPanel.add(pRed);
controlPanel.add(pGreen);
controlPanel.add(pBlue);
controlPanel.add(colorButton);
```

and the color panel can be constructed with the statement

```
colorPanel = new JPanel();
```

Next, the control panel and the color panel need to be added to the applet. We can use the `BorderLayout` manager and put the control panel in the north position and the color panel in the center position so that it fills the remaining space. The statements to do this are

```
Container cp = getContentPane();
cp.setLayout(new BorderLayout());
cp.add(controlPanel, "North");
cp.add(colorPanel, "Center");
```

Finally, the `init` method needs to add a listener to the button. We will do this using an inner class and the statement

```
colorButton.addActionListener(new ColorButtonHandler());
```

The inner class needs an `actionPerformed` method to read the three color codes from the input text fields and use them to set a new background color for the color panel. This class is given by

```
public class ColorButtonHandler implements ActionListener
{
   public void actionPerformed(ActionEvent e)
   {
      changeColor();
   }
}
```

where the `changeColor` method is given by

```
public void changeColor()
{
   int red = redField.getInt();
   int green = greenField.getInt();
   int blue = blueField.getInt();
   Color c = new Color(red, green, blue);
   colorPanel.setBackground(c);
   repaint();
}
```

The `repaint` method is important since it tells the event manager that it should request an update of the components on the screen to reflect the new background color. This has the effect of calling the `paintComponent` method of the panel.

Since an applet does not have a frame inside the browser window we can put a one pixel border rectangle around the applet panel. For applets this is not done using the `paintComponent` method but with the `paint` method. Therefore we include

```
public void paint(Graphics g)
{
   super.paint(g);
   Graphics2D g2D = (Graphics2D) g;
   Shape border = new Rectangle2D.Double(0,0,getWidth()-1, getHeight()-1);
   g2D.setPaint(Color.black);
   g2D.draw(border);
}
```

You can see this border in Figure 10.15 and Figure 10.16.

Finally, we need to read the three initial values for the red, green, and blue components from the parameter fields in the applet tag. This is done in the `init` method using

```
try // in case we run it as an application
{
   // try to get parameters from applet tag

   redValue = getParameter("redValue");
   greenValue = getParameter("greenValue");
   blueValue = getParameter("blueValue");

   // if they are null assign default values

   if (redValue == null) redValue = "125";
   if (greenValue == null) greenValue = "205";
   if (blueValue == null) blueValue = "125";
}
catch (NullPointerException e)
{
   redValue = "125";
   greenValue = "205";
   blueValue = "125";
}
```

The argument of the `getParameter` method specifies the parameter name as a string as defined by the name part of the parameter tag and the return value is the value of this parameter as specified by the value part of the parameter tag. The `try` block is necessary in case the applet tag has no parameter tags or in case it is run as an application using the `main` method.

This is all we need to write an applet. But if we also want to run the applet as a standalone application we need a `main` method that can create a frame and put our applet in it. This can be done as follows:

```
     public static void main(String[] args)
     {
        RGBColorApplet a = new RGBColorApplet();
        a.setSize(400,150);
        a.init();
        JFrame f = new JFrame();
        Container cp = f.getContentPane();
        cp.add(a);
        f.addWindowListener(new WindowCloser());
        f.setTitle("RGB colors");
        f.setSize(400,175);
        f.setVisible(true);
     }
```

We are in effect writing a “no-frills” appletviewer or browser here. First an applet object is constructed and initialized by calling its `init` method. Then a `JFrame` object called `f` is created and the applet object is added to its content pane.

Finally, we add a window listener, a title, a size for the frame, and we make the frame visible. For a pure applet class we do not need to worry about closing the frame since the applet viewer or the browser takes care of this. Here is the complete applet class:

Class `RGBColorApplet`

`book-project/chapter10/applets`

```
package chapter10.applets;
import custom_classes.InputJTextField;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * An applet for calculating RGB color values that can also be
 * run as an application.
 */
public class RGBColorApplet extends JApplet
{
   private JLabel redLabel;
   private JLabel greenLabel;
   private JLabel blueLabel;
   private InputJTextField redField;
   private InputJTextField greenField;
   private InputJTextField blueField;
   private JPanel controlPanel;
   private JPanel colorPanel;
   private JButton colorButton;

   // Applet parameters specified in applet tag
```

```
private String redValue, greenValue, blueValue;

public void init()
{
   try // in case we run it as an application
   {
      // try to get parameters from applet tag

      redValue = getParameter("redValue");
      greenValue = getParameter("greenValue");
      blueValue = getParameter("blueValue");

   // if they are null assign default values

      if (redValue == null) redValue = "125";
      if (greenValue == null) greenValue = "205";
      if (blueValue == null) blueValue = "125";
   }
   catch (NullPointerException e)
   {
      redValue = "125";
      greenValue = "205";
      blueValue = "125";
   }

   // Put red label and text field in panel
   // and set panel’s background color

   redLabel = new JLabel("Red", JLabel.CENTER);
   redLabel.setForeground(Color.black);
   redField = new InputJTextField(redValue, 4);
   JPanel pRed = new JPanel();
   pRed.add(redLabel);
   pRed.add(redField);
   pRed.setBackground(Color.red);

   // Similarly for a green label and text field

   greenLabel = new JLabel("Green", JLabel.CENTER);
   greenLabel.setForeground(Color.black);
   greenField = new InputJTextField(greenValue,4);
   JPanel pGreen = new JPanel();
   pGreen.add(greenLabel);
   pGreen.add(greenField);
   pGreen.setBackground(Color.green);

   // Similarly do a blue label and text field

   blueLabel = new JLabel("Blue", JLabel.CENTER);
   blueLabel.setForeground(Color.black);
   blueField = new InputJTextField(blueValue, 4);
   JPanel pBlue = new JPanel();
```

```
      pBlue.add(blueLabel);
      pBlue.add(blueField);
      pBlue.setBackground(Color.blue);

      colorButton = new JButton("Color");

      // Construct the control panel and add color panels to it

      controlPanel = new JPanel();
      controlPanel.setLayout(new FlowLayout());
      controlPanel.add(pRed);
      controlPanel.add(pGreen);
      controlPanel.add(pBlue);
      controlPanel.add(colorButton);

      // Construct a color panel

      colorPanel = new JPanel();

      // Add control panel and color panel to the applet’s content pane
      // using a border layout. The control panel is at the top of the
      // frame and the color panel fills the remainder of the frame.

      Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      cp.add(controlPanel, BorderLayout.NORTH);
      cp.add(colorPanel, BorderLayout.CENTER);

      colorButton.addActionListener(new ColorButtonHandler());
      changeColor();
   }

   /**
    *  Put a one pixel black rectangle around applet panel
    */
   public void paint(Graphics g)
   {
      super.paint(g);
      Graphics2D g2D = (Graphics2D) g;
      int xMax = getWidth() - 1;
      int yMax = getHeight() - 1;
      Shape border = new Rectangle2D.Double(0,0,xMax,yMax);
      g2D.setPaint(Color.black);
      g2D.draw(border);
   }

   /**
    * Inner class containing actionPerformed method to be called when
    * the color button in the control panel is clicked
    */
   public class ColorButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         changeColor();
      }
   }

   // Get RGB values, set background of the color panel to this color.

   private void changeColor()
   {
      int red = redField.getInt();
      int green = greenField.getInt();
      int blue = blueField.getInt();
      Color c = new Color(red, green, blue);
      colorPanel.setBackground(c);
      repaint();
   }

   public static void main(String[] args)
   {
      // Construct an applet object and initialize it

      RGBColorApplet a = new RGBColorApplet();
      a.setSize(400,150);
      a.init();

      // Construct frame and add applet to its content pane

      JFrame f = new JFrame();
      Container cp = f.getContentPane();
      cp.add(a);
      f.setTitle("RGB colors");
      f.setSize(400,175);
      f.setVisible(true);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

10.13.2 Running applets from BlueJ

It is easy to run and test applets from BlueJ. In the BlueJ project in Figure 10.17 the notation `<<applet>>` in the yellow class rectangle indicates that this class is an applet. When you right click on the rectangle the menu shown in Figure 10.18 appears.

To run the applet simply choose the “Run Applet” menu choice and the dialog box shown in Figure 10.19 appears. Here we can specify the applet tag. We have specified values for the three parameters and we have chosen a width of 400 pixels and a height of 200 pixels for the applet.

When you click OK BlueJ will create a small HTML file, `RGBColorApplet.html`, with the same name as the class having the contents

```
<html>
```

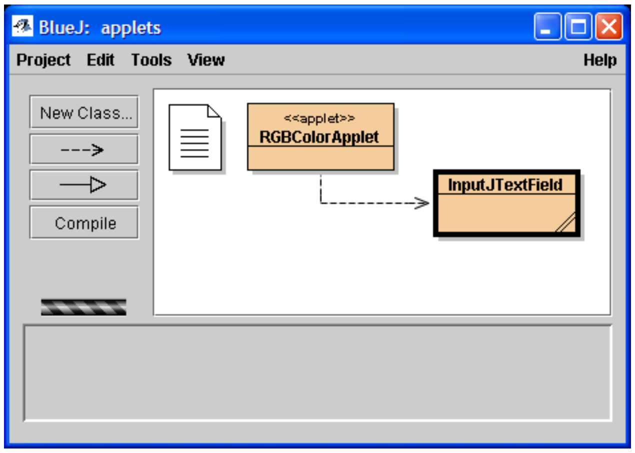


Figure 10.17: `RGBColorApplet` in a BlueJ project

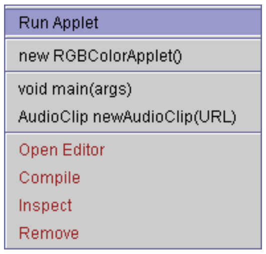
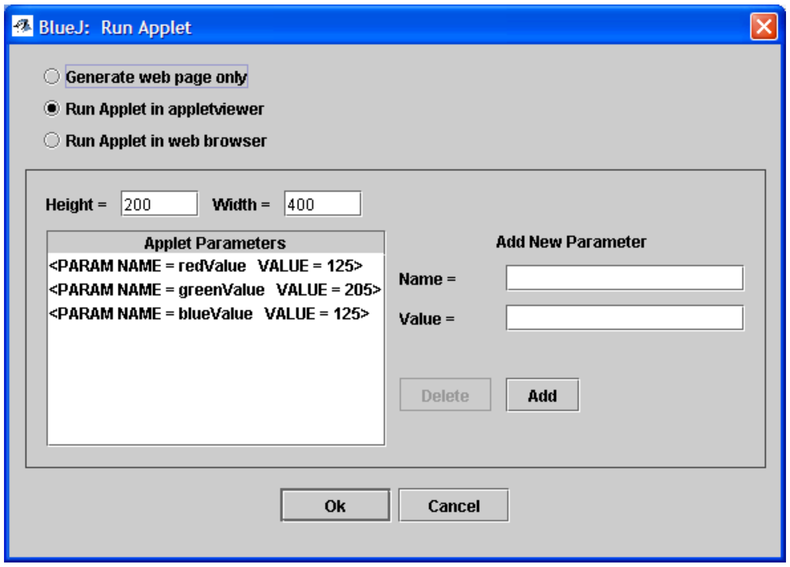


Figure 10.18: `RGBColorApplet` class menu



Generate web page only
[✓ Run Applet in appletviewer]
Run Applet in web browser

Height = 200 Width = 400

```
<PARAM NAME = redValue   VALUE = 125>
<PARAM NAME = greenValue   VALUE = 205>
<PARAM NAME = blueValue   VALUE = 125>
```

Figure 10.19: Defining `RGBColorApplet` applet parameters

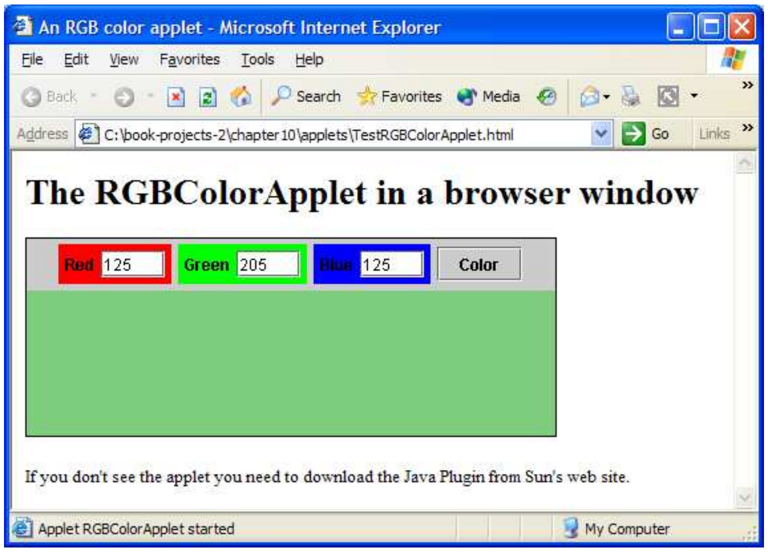
```
<!-- This file automatically generated by BlueJ Java Development  -->
<!-- Environment.  It is regenerated automatically each time the  -->
<!-- applet is run.  Any manual changes made to file will be lost -->
<!-- when the applet is next run inside BlueJ.  Save into a       -->
<!-- directory outside of the package directory if you want to    -->
<!-- preserve this file. -->
    <head>
        <title>RGBColorApplet Applet</title>
    </head>
    <body>
        <h1>RGBColorApplet Applet</h1>
        <hr>
        <applet code="RGBColorApplet.class"
            width=400
            height=200
            codebase="."
            alt="Your browser understands the &lt;APPLET&gt; tag
            but isn't running the applet, for some reason."
         >
   <PARAM NAME = redValue   VALUE = 125>
   <PARAM NAME = greenValue   VALUE = 205>
   <PARAM NAME = blueValue   VALUE = 125>

            Your browser is ignoring the &lt;APPLET&gt; tag!
        </applet>
        <hr>
    </body>
</html>
```

and runs the appletviewer using this file.

10.13.3 Running Java applets in a browser

The following HTML file shows how the applet tag can be used:

```
<html>
<head>
<title>An RGB color applet</title>
</head>

<body>

<h1>The RGBColorApplet in a browser window</h1>

<applet code = "RGBColorApplet.class" width="400" height="150">
<param name = "redValue"  value = "125">
<param name = "greenValue"  value = "205">
```


An RGB color applet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\book-projects-2\chapter10\applets\TestRGBColorApplet.html
Go Links
Red 125
Green 205
Blue 125
Color
Applet RGBColorApplet started
My Computer

The RGBColorApplet in a browser window

If you don't see the applet you need to download the Java Plugin from Sun's web site.

Figure 10.20: `RGBColorApplet` running in Internet Explorer

```
<param name = "blueValue"  value = "125">
</applet>

<p>
If you don’t see the applet you need to download the Java Plugin
from Sun’s web site.
</p>

</body>
</html>
```

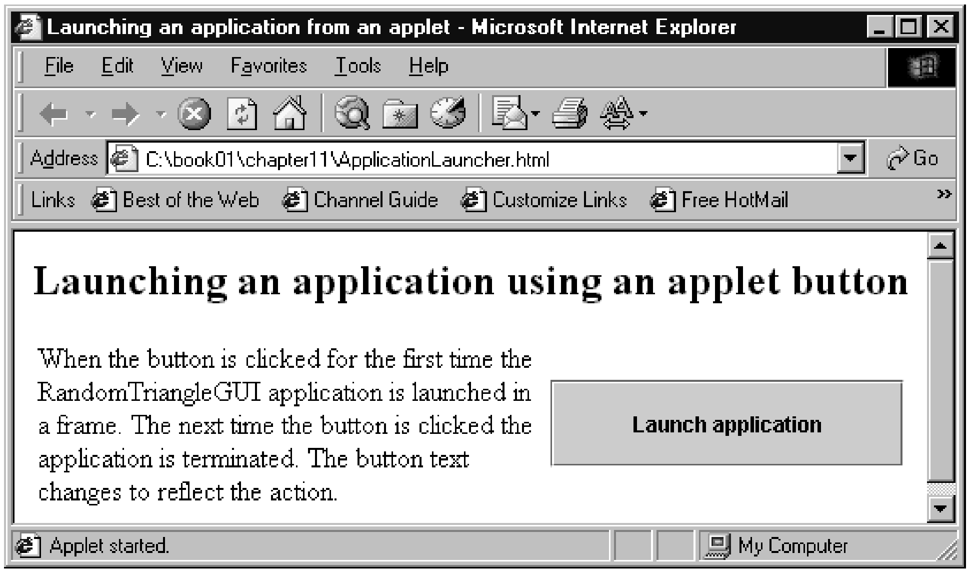
When you load this file into your browser the applet will start running as shown in Figure 10.20.

10.13.4 Launching Java applications from an applet

It is also possible to run a java application from an applet. As a simple example we write a small applet called `ApplicationLauncher` that contains only a button whose purpose is to launch the `RandomTrianglesGUI` application, considered earlier in this chapter, when the button is clicked. The application appears in a separate frame and can be closed and terminated by clicking the applet button again. The button will act like a toggle between launching the application and terminating the application. When the HTML page containing the button is replaced by a new page or when the browser is closed the application will also be terminated.

A picture of the applet running in Internet Explorer is shown in Figure 10.21 and the HTML file shown in the browser window is

```
<html>
<title>Launching an application from an applet</title>
```


Launching an application from an applet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address C:\book01\chapter11\ApplicationLauncher.html
Go
Links Best of the Web Channel Guide Customize Links Free HotMail
Launch application
Applet started.
My Computer

Launching an application using an applet button

When the button is clicked for the first time the RandomTriangleGUI application is launched in a frame. The next time the button is clicked the application is terminated. The button text changes to reflect the action.

Figure 10.21: `ApplicationLauncher` running in Internet Explorer

```
<body>
<h2>Launching an application using an applet button</h2>
<table>
<tr>
<td>
When the button is clicked for the first time the RandomTriangleGUI
application is launched in a frame. The next time the button is clicked
the application is terminated.
The button text changes to reflect the action.
</td>
<td>
<APPLET CODE = "ApplicationLauncher.class" WIDTH=200 HEIGHT=50>
</APPLET>
</td>
</tr>
</table>
</body>
</html>
```

Here is the applet class.

Class `ApplicationLauncher`

`book-project/chapter10/random_triangles`

```
package chapter10.random_triangles;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Show how to launch GUI application from a JApplet
 * using a button. When the button is clicked the frame appears.
 * When the button is clicked again the frame is destroyed.
 */
public class ApplicationLauncher extends JApplet implements ActionListener
{
   JFrame f;
   JButton launchButton;

   public void init()
   {
      f = null;
      launchButton = new JButton("Launch application");
      Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      cp.add(launchButton, "Center");
      launchButton.addActionListener(this);
   }

   /**
    * This method is called when browser terminates the applet.
    */
   public void destroy()
   {
      if (f != null) f.dispose();
   }

   /**
    * This method is called when the launcher button is clicked.
    * The button acts like a toggle.
    */
   public void actionPerformed(ActionEvent e)
   {
      if (f == null) // create application, display it
      {
         f = new RandomTrianglesGUI();
         f.setVisible(true);
         launchButton.setText("Terminate application");
      }
      else // dispose of frame to terminate application
      {
         f.dispose();
         f = null;
         launchButton.setText("Launch application");
      }
   }
}
```

The `init` method simply creates a button which fills the area allocated for the applet since the border layout is used. The `actionPerformed` method can detect whether the application object exists by testing its reference to see if it is `null`, in which case it creates the application in a `JFrame` and makes it visible. Otherwise the application is terminated. The text on the button is changed dynamically using the `setText` method to indicate what happens when the button is clicked. It is necessary to dispose of the application using the `dispose` method when the browser is closed or visits a new page.

10.14 Review exercises

► **Review Exercise 10.1** Define the following terms and give examples of each.

GUI	component	event listener
event-driven programming	text field	text area
listener list	`EventListener`	`ActionListener`
`ActionEvent`	`WindowListener`	`WindowEvent`
`JFrame`	`JComponent`	`JPanel`
`JLabel`	`JButton`	`JTextField`
`InputJTextField`	`JTextArea`	`JScrollPane`
layout manager	`FlowLayout`	`BorderLayout`
`GridLayout`	content pane	`addActionListener`
`addWindowListener`	inner class	`add`
`getSource`	`actionPerformed`	`SwingConstants`
`CloseableJFrame`	`Applet`	

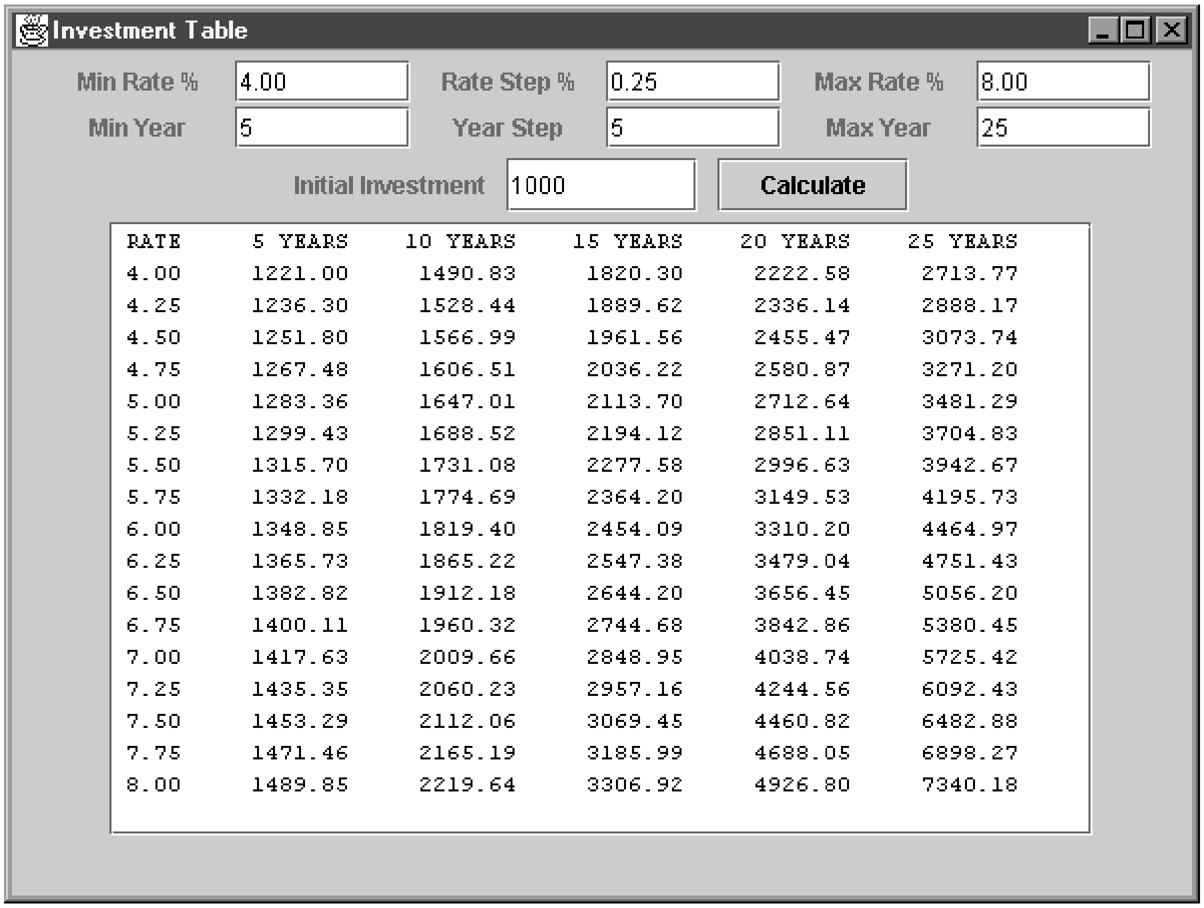
10.15 Programming exercises

► **Exercise 10.1 (GUI version of DoubleYourMoney)**
Write a GUI version of the doubling your money program (Chapter 7, page 323). Provide two `InputJTextField` objects for getting the initial investment and the annual interest rate in percent. Use a button that will perform the calculations when it is clicked and display the results in a scrollable `JTextArea` object.

► **Exercise 10.2 (A GUI version of InvestmentTable)**
We did a GUI version of the loan repayment program called `LoanRepaymentTableGUI`, page 553, using the reusable `LoanRepaymentTable` class which returned the table using `toString`.

Write a GUI version of the investment program using the `InvestmentTable` class, Chapter 7, page 353, that returns the table using `toString` Your application frame should look as close as possible to the frame shown in Figure 10.22.

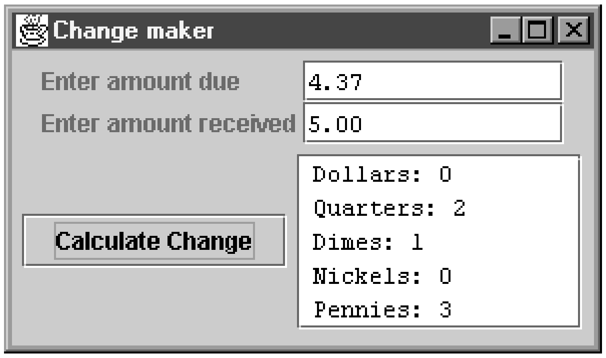
► **Exercise 10.3 (GUI version of ChangeMaker)**
Write a GUI version of a change maker program called `ChangeMakerGUI`. Your application frame



Min Rate % 4.00 Rate Step % 0.25 Max Rate % 8.00
Min Year 5 Year Step 5 Max Year 25
Initial Investment 1000

```
RATE     5 YEARS   10 YEARS   15 YEARS   20 YEARS   25 YEARS
4.00     1221.00    1490.83    1820.30    2222.58    2713.77
4.25     1236.30    1528.44    1889.62    2336.14    2888.17
4.50     1251.80    1566.99    1961.56    2455.47    3073.74
4.75     1267.48    1606.51    2036.22    2580.87    3271.20
5.00     1283.36    1647.01    2113.70    2712.64    3481.29
5.25     1299.43    1688.52    2194.12    2851.11    3704.83
5.50     1315.70    1731.08    2277.58    2996.63    3942.67
5.75     1332.18    1774.69    2364.20    3149.53    4195.73
6.00     1348.85    1819.40    2454.09    3310.20    4464.97
6.25     1365.73    1865.22    2547.38    3479.04    4751.43
6.50     1382.82    1912.18    2644.20    3656.45    5056.20
6.75     1400.11    1960.32    2744.68    3842.86    5380.45
7.00     1417.63    2009.66    2848.95    4038.74    5725.42
7.25     1435.35    2060.23    2957.16    4244.56    6092.43
7.50     1453.29    2112.06    3069.45    4460.82    6482.88
7.75     1471.46    2165.19    3185.99    4688.05    6898.27
8.00     1489.85    2219.64    3306.92    4926.80    7340.18
```

Figure 10.22: `InvestmentTable` application



Enter amount due 4.37
Enter amount received 5.00

```
Dollars: 0
Quarters: 2
Dimes: 1
Nickels: 0
Pennies: 3
```

Figure 10.23: `ChangeMaker` application

should look as close as possible to the frame shown in Figure 10.23. The checkout person enters the amount due and the amount received by the customer. When the calculate button is pressed the change is displayed as shown in the text area. Use an inner class to implement the `ActionListener` interface.

► Exercise 10.4 (A better GUI version of ChangeMaker)
Make a better version of `ChangeMakerGUI` that does not display zero values. For example, for the values shown in Figure 10.23 the dollars and nickels rows would not be shown at all. Also this version should check the input carefully: amounts entered should be non-negative and the amount received should be greater or equal to the amount due. Display appropriate messages in the text area for illegal input.

► Exercise 10.5 (A modification to RGBColorApplet)
Modify RGBColorApplet so that pressing the enter key in any of the three text fields also triggers a repaint of the color panel.

► Exercise 10.6 A GUI application version of RGBColorApplet)
The `RGBColorApplet` class was written to work both as an applet and a GUI application. Write a version called `RGBColorGUI` that works only as an application.

► Exercise 10.7 (An applet version of InvestmentTableGUI)
Write an applet version of `InvestmentTableGUI` called `InvestmentTableApplet` and write an HTML file for it using `HTMLConverter` so that it will run in a browser (you can modify the file `RGBColorApplet.html` to use `InvestmentTableApplet.class` and change the width and height values).

► Exercise 10.8 (Launching applications using applet buttons)
Using `ApplicationLauncher` as a guide, page 588, write an applet called `ProgramLauncher` that contains several buttons. Each button should launch an application class in a frame. Use some of the applications in this chapter.

Chapter 11

Files and Streams

Data processing with sequential files

Outline

File concepts

Reading and writing sequential files

Binary and text file structures

`InputStream` and `OutputStream` hierarchies

`Reader` and `Writer` hierarchies

File I/O error handling using exceptions

Reading and writing byte streams

Reading and writing character streams

`FileCopier` and `FileSearcher` programs

`ByteViewer` program for viewing bytes in a file

Text files of `BankAccount` objects

`ArrayList` class

Binary object files of `BankAccount` objects

11.1 Introduction

In this chapter we introduce files and streams and some applications that use them. Streams are sequences of data items that can be connected to a file. First, a detailed survey of some of the built-in stream and file classes is given. These are low-level classes for processing streams of bytes and characters. We also show how to do higher-level text file processing using `BankAccount` objects as a standard example. Our approach is to use inheritance and extend the built-in classes to read and write `BankAccount` objects. Polymorphic binary object files for reading and writing serialized objects are introduced using the `BankAccount` and `JointBankAccount` classes as examples.

11.2 File concepts

11.2.1 What is a file?

A file is an organized collection of data on an external storage device such as a disk (floppy disk, hard disk, CD-ROM). It is referenced by its name. Files are independent of the programs that create and manipulate them as opposed to data stored in the main computer memory (RAM memory), which is available only while the program is running. This data is lost when the program exits or when the computer is turned off. Thus, files provide long term data storage. Also, their size can be larger than the available memory.

For example, if a student marks program creates an array of student marks for the assignments and tests in a course then this array is created in memory and will be lost when the program exits. Since it is important to keep this array for a longer time, it is necessary to write it to a file. The marks program can then read the file at a later time for further processing such as correcting errors, adding new marks, or computing averages.

11.2.2 File organization on a sisk

In order to work with files it is helpful to have some understanding of how they are organized on a disk and how to communicate with the operating system (OS) in order to read and write them.

Computer memory is normally organized as a large sequential collection of bytes, each of which has an address. On the other hand each surface of a disk is organized into concentric tracks, and each track is composed of a number of sectors, each of which can hold a fixed number of bytes. Each file occupies a certain number of sectors which are not necessarily consecutive. In order for the OS to keep track of each file stored on a disk it keeps a file allocation table (FAT) on the disk. There is an entry in the FAT for each file that gives information such as the name of the file, the size of the file, and the location on the disk (track and sector number) of the first sector allocated to the file. Each sector contains a reference to the next sector (link to the next sector) so the complete file can be accessed by following these links. New files can be written to a disk by locating unused sectors to store the data and linking them together.

11.2.3 Efficient file access using buffering

Another important consideration when using files is that accessing data on a disk is thousands of times slower than accessing data stored in main memory (milliseconds versus nanoseconds).

Since file access times are much greater than memory access times it is important that files are accessed efficiently. For each file that is currently being read or written the operating system maintains information about the file in a data structure called a file control block (FCB). The FCB resides in memory and contains a data buffer to hold the next block of bytes from the file and all information necessary to communicate with the file (e.g., file name, size of file in bytes, file access rights). Fortunately we do not need to know the detailed structure of an FCB since high level languages provide convenient functions for working with files at a high level. The OS is responsible for the lower level file operations and the FCB.

The OS does not read or write single bytes at a time. Instead, it is much more efficient to use the data buffer associated with the FCB to read or write one data buffer at a time. For example, if the size of the data buffer is 4096 bytes, then that many bytes are read or written at a time. This method of doing file input/output (I/O) one data buffer at a time, rather than one byte at a time, is called buffered I/O.

11.2.4 File access rights

Other properties of a file are its access rights. Unless you are writing an OS, your application programs do not directly access files on the disk. Instead, they make higher level I/O requests to the OS. This gives the OS a chance to do the I/O efficiently and also check that the application program has permission to access the file. Each file created by the OS is given a set of permissions to indicate who has read access and write access to the file. Some files have read-only access and others have read/write access.

11.2.5 Reading and writing sequential files

A **sequential file** is a sequence of data items stored on a disk that can only be accessed one item at a time sequentially, from the first item to the last item, in the order in which the items were written to the file.

Reading a sequential file

Reading a sequential file means to transfer data items from the file to your application program as follows:

(1) The program makes a request to the OS to open a file for reading.

(2) If the program has permission to read the file the OS allocates an FCB containing a data buffer. This is called “opening the file for reading”.

(3) The program makes a read request to the OS for a data item.

(4) If the data buffer is empty the OS fills it with data from the file if there is more data to read and transfers data to the program from the buffer. If end of file (EOF) has been reached the OS sends an EOF message to the program indicating that there is no more data to read.

(5) If the program wants to read more data items steps (3) and (4) are repeated, otherwise the program should close the file (deallocate the FCB for the file).

This gives the following simple pseudo-code model for reading a sequential file:

```
Open the file for reading.
Read a data item from the file.
WHILE not at end of file DO
    Process the data item.
    Read the next data item.
END WHILE
Close the file
```

Your program may request data one byte at a time but this does not mean that the disk is accessed for each byte. Instead, if the buffer is not empty then the byte will be given to your program from the buffer, a much faster memory-to-memory transfer.

Also, if your program could only request that data be read one byte a a time, it would still be inefficient for your program to read a large file in a big loop and we will see that an extra level of input buffering is usually available to application programs.

Writing a sequential file

Similarly, writing a sequential file means to transfer data items from your application program to the file as follows:

(1) The program makes a request to the OS to open a file for writing.

(2) If the program has permission to write the file the OS allocates an FCB containing a data buffer. This is called “opening the file for writing”. Normally, this erases a file that already exists although it is possible to indicate that you want to append data to an existing file rather than erase it and create a new file with the same name.

(3) The program makes a write request to the OS and provides a data item to be written.

(4) The OS transfers data from the program to the data buffer. If the buffer becomes full, the OS writes the entire buffer to the disk and empties the buffer in preparation for more write requests.

(5) If the program wants to write more data items steps (3) and (4) are repeated, otherwise the program should close the file (deallocate the FCB for the file). Normally, closing a file writes to the disk any data left in the buffer. This is called flushing the buffer. Sometimes a separate flush operation is provided by the OS. Thus, it is important to close files so that all data is written to the file.

This gives the following simple pseudo-code model for writing a sequential file:

```
Open the file for writing.
WHILE there are data items to write DO
    Write the next data item.
END WHILE
Close the file
```

As for reading, if your program could only request that data be written one byte a a time, it would still be inefficient for your program to write a large file in a loop and we will see that an extra level of output buffering is usually available to application programs.

11.2.6 File access methods

At the lowest level files are collections of bytes. These bytes may be organized into higher level structures. For example, a database file consists of a number of records, each composed of a number of fields, and each field is composed of a number of bytes. The two most important kinds of file access are sequential access and random (direct) access.

Sequential access files

As mentioned above a sequential file is accessed one item at a time, in the order in which the items are written (for example, an item may be a byte, an encoded Unicode character, or a database record). If the item is a byte we say that the file is a sequential file of bytes.

The fundamental read operation available for a sequential file open for input is to read the next item from the file. Similarly, the fundamental write operation for a sequential file open for output is to write the next item to the file. A sequential file can be opened either for reading or for writing but not both at the same time. For example, to switch from writing to reading it is necessary to first close the output file and then open it again as an input file for reading from the beginning.

To read the n-th item from a sequential file it is always necessary to read the first $n-1$ items. This is a natural structure for many kinds of files. For example, if you want to read a file one item at a time and write a new file containing only the lines of the file having a given search pattern, then a sequential file is appropriate. Here the items are the lines in the file so we can read lines one at a time and if the pattern is present write the line to the output file.

Random access files

On the other hand, if it is necessary to make many modifications (e.g., insertions, deletions, updates) to random records in a file the sequential access method is very inefficient. For example, to make a new version of a file that has an updated version of item number 100, it is necessary to read the first 99 items from the input file, copy them to an output file, read item 100, modify it, write item 100 to the output file, and then write the remaining items of the input file to the output file. This may be fine for modifying a few items that occur in a known sequence but not for modification of random items in an on-line airline reservation system or an account database for a bank.

For this reason there is a kind of file called a **random access** or **direct access** file that is organized as a file of fixed size records. Each record must have a fixed size so that its position in the file can be calculated as a multiple of the record size. The file can be open for both reading and writing at the same time and each record can be read or written simply by referring to its record number in the file. For example, to update record 100 it is only necessary to read record 100, modify it and write it back to the file in the place it was read from. Random access files are like arrays where any item can be accessed by specifying its array index.

Java has support for both sequential and random access files but only sequential files will be considered in this chapter.

11.3 File structure

Every file at the lowest level is just a sequence of bytes. Usually this sequence has a structure imposed on it and a meaning determined by the program that wrote the file in the first place.

For example, files are often classified as **text files** or **binary files**. As opposed to a binary file, a text file is meant to be “human readable”. When you use a standard text editor, as opposed to a word processor, you are working with a text file. Binary files cannot be created or edited with these editors.

11.3.1 Text file structure

For the English language the bytes of a text file are commonly interpreted as 7-bit ASCII codes (0 to 127) for the various control, alphanumeric, and punctuation characters. For many languages there are accented characters so this 7-bit code is extended to the 8-bit LATIN-1 code. Some of the extra codes in the range 128 to 255 are used to denote letters with accents.

Since text files are files of lines, one or more special bytes are needed to indicate where each line ends. In Unix this is the new line character (code 10, denoted by `\n` in Java). For the Mac OS this character is carriage return (code 13, denoted by `\r` in Java) and for Windows 95/98/XP a carriage return followed by a line feed is used. Thus, text files are files of codes corresponding to the readable characters with occasional end of line indicators.

However, for many other languages 8-bit codes are not big enough, so the Unicode standard was created to extend the ASCII and LATIN-1 codes. Unicode characters use 16-bit codes. The first 128 characters are the 7-bit ASCII codes and the next 128 codes are the extension to LATIN-1. This leaves a lot of codes for other languages. In Java a text file is called a character file and it uses an encoding for a Unicode character set. For example, ASCII and LATIN-1 can be encoded using a single byte per character instead of two but the Chinese languages are pictographic and would require two bytes per character.

We will see that Java has I/O classes for reading or writing text files that contain encoded Unicode characters. The `char` and `String` data types represent characters internally using Unicode. That’s why the `char` type is a 16-bit data type, whereas for most older computer languages it is an 8-bit type.

11.3.2 Binary file structure

Although it is common to think of a text file as a special kind of binary file we prefer to use the term "binary file" to refer to a file that is not a text file. For example, the byte-code files produced by the Java compiler are binary files. If you try to load them into a text editor you will see gibberish. There are many other kinds of binary files. We will see that Java has I/O classes for reading and writing binary files, the primitive data types in binary form, and even classes for reading and writing serialized objects in binary form.

■ $\boxed{\text{EXAMPLE 11.1}}$ **(Binary and text representations of an integer)** To illustrate the difference between text and binary files let us look at how the `int` value 123526 is stored. Internally, since an `int` is 32-bits, it is stored as the four bytes (expressed in binary)

```
00000000 00000001 11100010 10000110
```

which represent $0 \times 256^3 + 1 \times 256^2 + 226 \times 256^1 + 134 \times 256^0$. In a binary file it would also be stored this way.

However, in a text file it would be stored as the character string `"123526"`. If the text file used ASCII codes then the stored value would occupy six bytes, one for each of the ASCII codes. The ASCII codes for "0" to "9" in decimal are 48 to 57 so the number is stored as the 6 bytes (expressed in binary: 31,32,33,35,32,36 hex)

```
00110001 00110010 00110011 00110101 00110010 00110110
```

These are very different representations: in a binary file every `int` value occupies 4 bytes but as a character string the number of bytes varies from 1 byte for single character numbers like `"1"` all the way to 10 bytes for the largest `int` number string `"2147483647"`. Internally, in the `char` and `String` data types, characters are represented by Unicode so each character in the string `"123526"` would occupy two bytes. For the ASCII code the first byte would be zero. This gives the full 12-byte Unicode representation

```
00000000 00110001 00000000 00110010 00000000 00110011
00000000 00110101 00000000 00110010 00000000 00110110
```

where the first byte of each 16-bit character is zero. Since the first byte is always 0 for the Unicode representation of the ASCII code it would be wasteful to use 12 bytes to store the numeric string instead of 6 bytes. Therefore, Unicode ASCII characters are normally stored in files using a simple single-byte encoding per character by dropping the zero bytes. For languages with more than 256 characters a more complicated encoding would be needed. ■

11.3.3 Streams

A **stream** is a sequence of items such as bytes or characters. If the items are bytes the stream is called a **byte stream** and if they are characters it is called a **character stream**. Streams are more general than files in the sense that a stream doesn't need to be connected to a file. Streams are objects in Java. File concepts such as opening a file for reading, reading from a file, opening a file for writing, writing to a file, closing a file, flushing a file, binary file and text file, also apply to streams. If a stream is connected to a file it is called a **file stream**.

Input streams are objects that act as a source or input for a sequence of items that can be input one item at a time using read methods. Similarly, output streams are objects that act as as sink or output for a sequence of items that can be output one item at a time using write methods.

Streams can also be connected together. For example, the output of one stream can be used as the input to another stream. Thus, a sequential file is represented simply as a stream connected to a file at one end: the source for an input stream is a file and the sink for an output stream is a file.

11.4 Java stream and file I/O class hierarchies

Java has two class hierarchies for byte stream I/O. The abstract `InputStream` class is at the top of the input hierarchy and the abstract `OutputStream` class is at the top of the output hierarchy. The classes in these hierarchies are shown in Figure 11.1. Each class name has the word `Stream` in it. These classes are used to do I/O with byte streams without any interpretation of the bytes. `FileInputStream` and `FileOutputStream` are the only file stream classes. They are used to connect other streams to files to provide file input and output, respectively.

Similarly, there are two class hierarchies for character (text) stream I/O. The abstract `Reader` class is at the top of the input hierarchy and the abstract `Writer` class is at the top of the output hierarchy. The classes in these hierarchies are shown in Figure 11.2. Each class name has the word `Reader` or `Writer` in it. These classes are used for doing I/O with character streams using a Unicode encoding (output) or decoding (input). The `FileReader` and `FileWriter` classes can be used to connect character streams to files to provide text file input and output, respectively.

Two of the classes also have the word stream in their name: `InputStreamReader` is used to read a byte stream and decode it as a character stream and `OutputStreamWriter` is used to write a character stream and encode it as a byte stream. These two classes are needed in order to work with character streams since at the lowest level files in Java are byte streams. We will see that these converter classes are not normally used explicitly since the `FileReader` and `FileWriter` convenience classes use them internally to do the conversion.

Since streams and files are objects in Java, we first need to use the class constructors to construct and open them. We will see that some of the class constructors and methods can throw exceptions from the `IOException` class hierarchy if the opening of the stream or file fails, or a read or write operation fails.

11.4.1 `InputStream` hierarchy

Each class in the `InputStream` hierarchy has methods for reading bytes from a byte stream. Only `FileInputStream` is associated with a file. It is the only class in the hierarchy whose constructors can specify a file as an argument. Most of the stream classes have an `InputStream` object as an argument so the various streams in this hierarchy can be connected together to perform byte stream input.

The most important classes in the `InputStream` hierarchy are the `FileInputStream` class and the `BufferedInputStream` class.

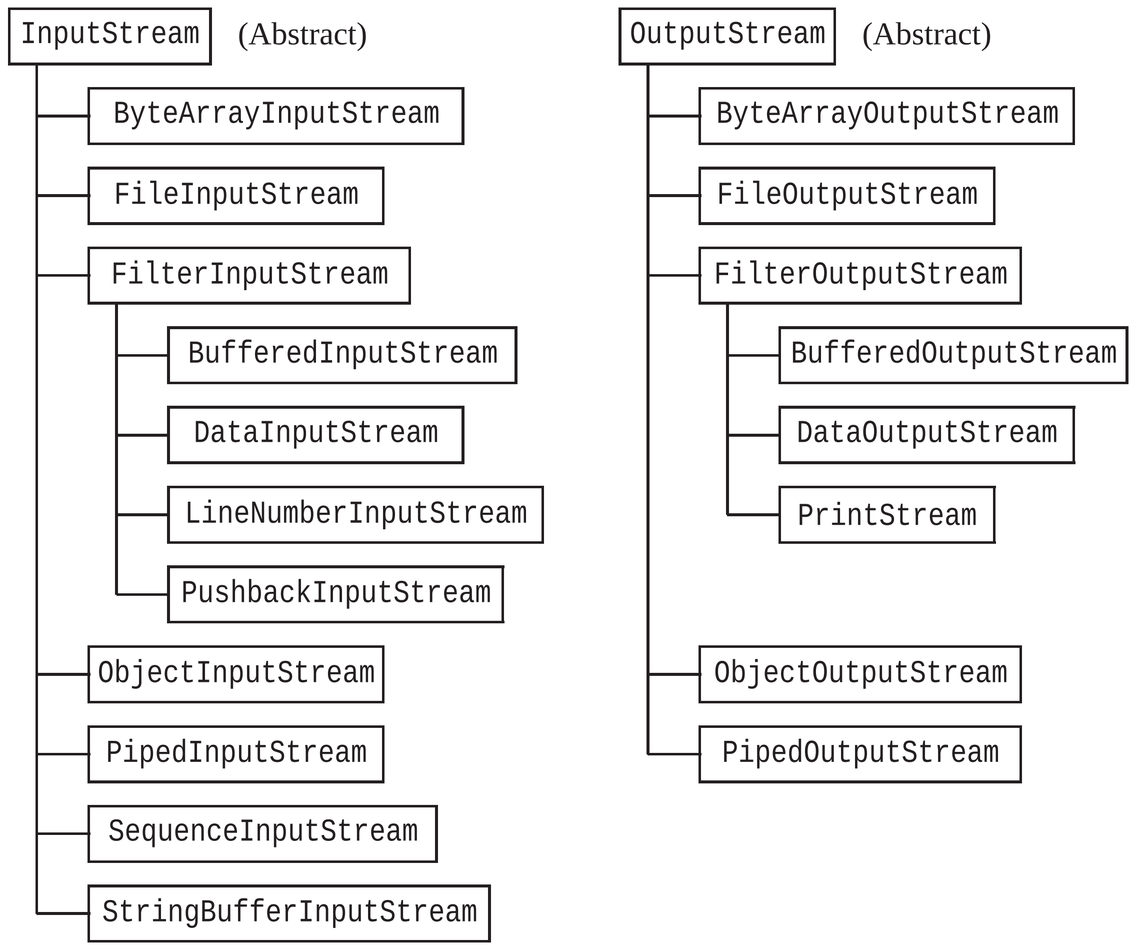


Figure 11.1: The `InputStream` and `OutputStream` hierarchies for byte stream I/O. All classes are in the `java.io` package and each class extends from the abstract `InputStream` or `OutputStream` class

■ EXAMPLE 11.2 **(Opening an input file as a byte stream)** `FileInputStream` has a constructor with prototype

```
public FileInputStream(String fileName) throws FileNotFoundException
```

A new feature called a `throws` clause is present at the end of this prototype and will be explained in more detail later. Here it means that when the constructor is called to create (open) a file for reading it will throw an exception of type `FileNotFoundException`, a subclass of `IOException`, if there is an error finding or opening the file. The declaration

```
FileInputStream fin = new FileInputStream("inFile.dat");
```

will construct a file object called `fin` and open a file called `inFile.dat` for reading. The `fin` object can now be used to read a stream of bytes from the file. ■

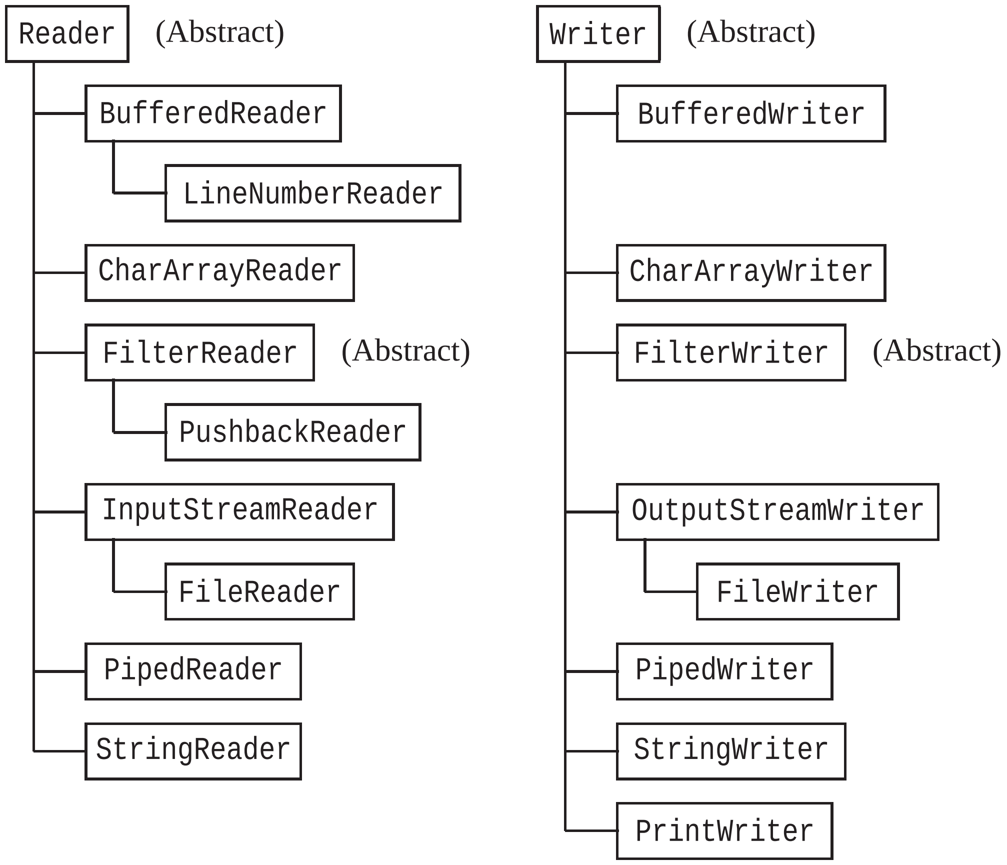


Figure 11.2: The `Reader` and `Writer` hierarchies for character stream I/O. All classes are in the `java.io` package and each class extends from the abstract `Reader` or `Writer` class

The `FileInputStream` class does not provide for buffering at the application program level so it can result in inefficient input unless buffering is added. This can be done by connecting it to a `BufferedInputStream`, as shown in the following example.

■ EXAMPLE 11.3 **(Buffering an input byte stream)** `BufferedInputStream` has a constructor with prototype

```
public BufferedInputStream(InputStream in)
```

This constructor has an `InputStream` as an argument so it can be used to buffer any stream that is a subclass. In particular a `FileInputStream` "is a" `InputStream` so we can provide buffered file input using

```
FileInputStream fin = new FileInputStream("inFile.dat");
BufferedInputStream in = new BufferedInputStream(fin);
```

Normally, it is not necessary to reference the `fin` variable anymore so it is common to use anonymous objects and combine these two declarations:

```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream("inFile.dat"));
```

The `in` object can now be used to read a buffered stream of bytes from the file. ■

11.4.2 `OutputStream` hierarchy

Each class in the `OutputStream` hierarchy has methods for writing bytes to a byte stream. Only `FileOutputStream` is associated with a file. It is the only class in the hierarchy whose constructors can specify a file argument. Most of the stream classes have an `OutputStream` object as an argument so the various streams in this hierarchy can be connected together to perform byte stream output.

The most important classes in the `OutputStream` hierarchy are the `FileOutputStream` class and the `BufferedOutputStream` class.

■ EXAMPLE 11.4 **(Opening an output file as a byte stream)** `FileOutputStream` has a constructor with prototype

```
public FileOutputStream(String fileName) throws FileNotFoundException
```

The declaration

```
FileOutputStream fout = new FileOutputStream("outFile.dat");
```

will construct a file object called `fout` and open a file called `outFile.dat` for writing. The `fout` object can now be used to write a stream of bytes to the file. ■

The `FileOutputStream` class does not provide for buffering at the application program level so it can provide inefficient output unless buffering is added. This can be done by connecting it to a `BufferedOutputStream`, as shown in the following example.

■ EXAMPLE 11.5 **(Buffering an output byte stream)** `BufferedOutputStream` has a constructor with prototype

```
public BufferedOutputStream(OutputStream out)
```

This constructor has an `OutputStream` as an argument so it can be used to buffer any stream that is a subclass. In particular a `FileOutputStream` "is a" `OutputStream` so we can provide buffered file output using

```
FileOutputStream fout = new FileOutputStream("outFile.dat");
BufferedOutputStream out = new BufferedOutputStream(fout);
```

Normally, it is not necessary to reference the `fout` variable anymore so it is common to use anonymous objects and combine these two declarations:

```
BufferedOutputStream out =
   new BufferedOutputStream(new FileOutputStream("outFile.dat"));
```

The `out` object can now be used to write a buffered stream of bytes to the file. ■

11.4.3 `Reader` hierarchy

Each class in this hierarchy has methods for reading characters from a character stream or file. In the `Reader` hierarchy the `InputStreamReader` and `FileReader` classes are associated with files. The other classes are associated with streams and can take a `Reader` object as an argument so the various streams in this hierarchy can be connected together to perform character stream input.

Since every file is a byte stream at the lowest level it is first necessary to use an object from the `FileInputStream` class to connect to the file. This defines a byte stream which must be connected to an `InputStreamReader` to do the decoding from bytes to characters. This class assumes a default encoding was used to write the file, although there is a constructor that can specify another encoding.

■ EXAMPLE 11.6 **(Opening an input file as a character stream)** `InputStreamReader` has a constructor with prototype

```
public InputStreamReader(InputStream in)
```

so the declaration

```
InputStreamReader in =
   new InputStreamReader(new FileInputStream("inFile.dat"));
```

will construct a character file stream object called `in` and open a file called `inFile.dat` for reading.

Since this is such a common operation when working with character streams a convenience class called `FileReader` is provided to do this connection. It's constructor has the prototype

```
public FileReader(String fileName) throws FileNotFoundException
```

Using this class we can construct a character stream input file using the simpler declaration

```
FileReader in = new FileReader("inFile.dat");
```

In either case the `in` object can be used to read a character stream from the file. ■

■ EXAMPLE 11.7 **(Buffering an input character stream)** `BufferedReader` has a constructor with prototype

```
public BufferedReader(Reader in)
```

This constructor has a `Reader` as an argument so it can be used to buffer any stream that is a subclass. In particular a `FileReader` "is a" `Reader` so we can provide buffered input using

```
FileReader fin = new FileReader("inFile.dat");
BufferedReader in = new BufferedReader(fin);
```

Normally it is not necessary to reference the `fin` variable anymore so it is common to use anonymous objects and combine these two declarations:

```
BufferedReader in =
   new BufferedReader(
      new InputStreamReader(
         new FileInputStream("inFile.dat")));
```

or equivalently, using the `FileReader` convenience class

```
BufferedReader in = new BufferedReader(new FileReader("inFile.dat"));
```

The `in` object can now be used to read a buffered character stream from the file. ■

11.4.4 `Writer` hierarchy

Each class in this hierarchy has methods for writing characters to a character stream or file. In the `Writer` hierarchy the `OutputStreamWriter` and `FileWriter` classes are associated with files. The other classes are associated with streams and can take a `Writer` object as an argument so the various streams in this hierarchy can be connected together to perform character stream output.

Since every file is a byte stream at the lowest level it is first necessary to use an object from the `FileOutputStream` class to connect to the file. This defines a byte stream which must be connected to an `OutputStreamWriter` to do the encoding from characters to bytes. This class assumes a default encoding is used to write the file, although there is a constructor that can specify another encoding.

■ EXAMPLE 11.8 **(Opening an output file as a character stream)** `OutputStreamWriter` has a constructor with prototype

```
public OutputStreamWriter(OutputStream out)
```

so the declaration

```
OutputStreamWriter out =
   new OutputStreamWriter(new FileOutputStream("outFile.dat"));
```

will construct a file stream object called `out` and open a file called `outFile.dat` for writing.

Since this is such a common operation when working with character streams a convenience class called `FileWriter` is provided to do this connection. It's constructor has the prototype

```
public FileWriter(String fileName) throws IOException
```

Using this class we can construct a character stream output file using the simpler declaration

```
FileWriter out = new FileWriter("outFile.dat");
```

In either case the `out` object can now be used to write a character stream to the file. ■

■ EXAMPLE 11.9 **(Buffering an output character stream)** `BufferedWriter` has a constructor with prototype

```
public BufferedWriter(Writer out)
```

This constructor has a `Writer` as an argument so it can be used to buffer any stream that is a subclass. Therefore a buffered output character stream object, `out`, connected to the file `outFile.dat` can be constructed using

```
FileOutputStream fout = new FileOutputStream("outFile.dat");
OutputStreamWriter s = new OutputStreamWriter(fout);
BufferedWriter out = new BufferedWriter(s);
```

Normally it is not necessary to reference the intermediate stream variables `fout` and `s` so it is common to use anonymous objects and write

```
BufferedWriter out =
   new BufferedWriter(
      new OutputStreamWriter(
         new FileOutputStream("outFile.dat")));
```

or equivalently, using the `FileWriter` convenience class

```
BufferedWriter out = new BufferedWriter(new FileWriter("outFile.dat"));
```

In either case the `out` object can be used to write a buffered character stream to the file. ■

11.4.5 `File` class

An object of the `File` class in `java.io` can be used to represent a file. It can be used as an argument to constructors that take a file name as a string. For example, we can define a `File` object from a given file name using

```
File outFile = new File("outFile.dat");
```

and then use

```
BufferedWriter out = new BufferedWriter(new FileWriter(outFile));
```

The `File` class is useful because it has methods that can determine whether a file exists, delete a file, or rename a file, for example.

11.4.6 Summary

In summary, to open a byte stream for buffered reading or writing we can use

```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream(inFile));
BufferedOutputStream out =
   new BufferedOutputStream(new FileOutputStream(outFile));
```

and to open a character stream for buffered reading or writing we can use

```
BufferedReader in = new BufferedReader(new FileReader(inFile));
BufferedWriter out = new BufferedWriter(new FileWriter(outFile));
```

Here `inFile` and `outFile` can be either strings representing the file names or `File` objects.

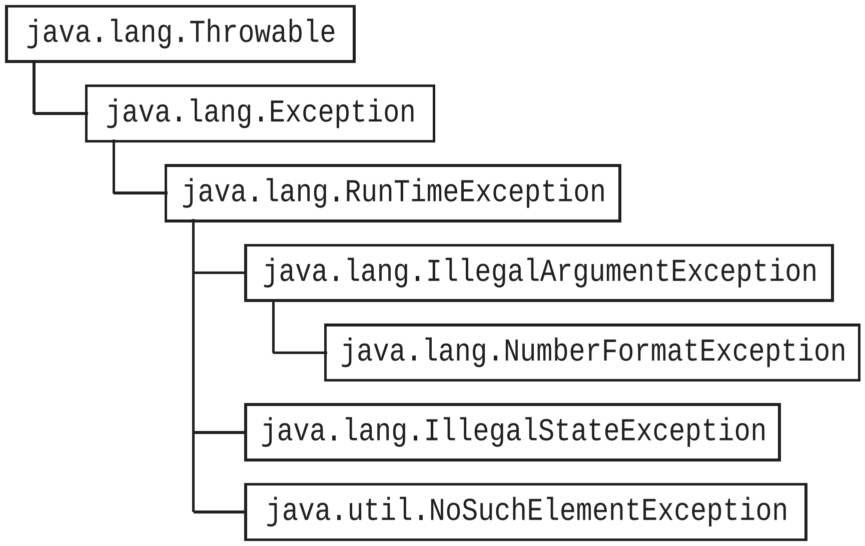

java.lang.Throwable
java.lang.Exception
java.lang.RunTimeException
java.lang.IllegalArgumentException
java.lang.NumberFormatException
java.lang.IllegalStateException
java.util.NoSuchElementException

Figure 11.3: Part of the unchecked exception sub-hierarchy of `RunTimeException`.

11.5 File I/O error handling using exceptions

11.5.1 Unchecked exceptions

We have used exceptions such as `IllegalArgumentException` and `NumberFormatException` before and we have shown how to throw them, or catch them using a `try-catch` block. Exceptions like these are called **unchecked exceptions** and are subclasses of `RunTimeException`. Part of this exception hierarchy is shown in Figure 11.3. It is not necessary to catch unchecked exceptions. The compiler doesn't check whether they are caught or not. If they are not caught the Java interpreter will do it at run-time. That's why they are subclasses of `RunTimeException`.

11.5.2 Checked exceptions

Other kinds of exceptions are called **checked exceptions**. In particular most of the methods in the I/O classes can throw an exception of type `IOException` or one of its subclasses. Part of this exception hierarchy is shown in Figure 11.4. For a checked exception it is necessary either to catch the exception or to indicate that it can be thrown in what is called the `throws` clause of a method or constructor. Failure to do this results in a compiler error: the compiler checks so that's why they are called checked exceptions.

■ EXAMPLE 11.10 **(The `throws` clause in a constructor)** One of the `FileInputStream` class constructors has the prototype

```
public FileInputStream(String fileName) throws FileNotFoundException
```

This prototype explicitly indicates, using a `throws` clause at the end of the prototype, that a `FileNotFoundException` may be thrown when the constructor is called to create (open) a file for reading. ■

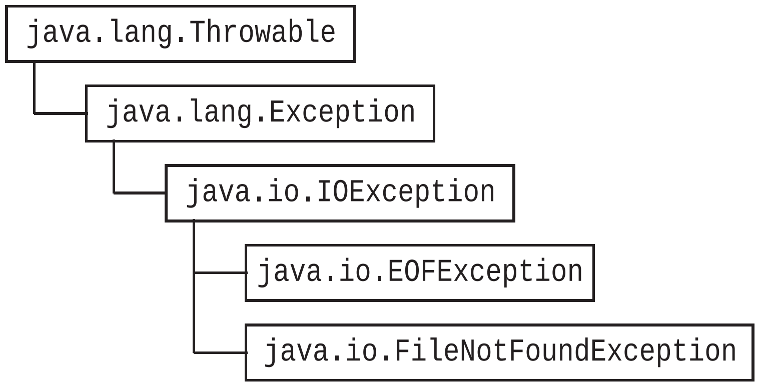


Figure 11.4: Part of the checked exception sub-hierarchy of `IOException`.

The Java documentation always tell you whether a method can throw a checked exception or not. It is not necessary to include a `throws` clause for unchecked exceptions, but if you are writing a method that can throw a checked exception and you don't catch it and you omit the `throws` clause the compiler will give you a nice error message telling you that you should either catch the exception or indicate it in the throws clause. Of course if you do catch the exception then the `throws` clause is not used.

11.6 Reading and writing byte streams

When a byte stream is opened for reading using a stream constructor, there are various read methods for reading bytes sequentially from the stream. If the stream is connected to a file then these methods will read bytes from the file. Similarly, when a byte stream is opened for writing there are various write methods for writing bytes sequentially to the stream and if the stream is connected to a file these methods will write bytes to the file.

11.6.1 Reading bytes from an input byte stream

The abstract `InputStream` class declares three methods for reading bytes from a stream and a method for closing a stream. The prototypes are

```
public abstract int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int startPos, int numBytes) throws IOException
public void close() throws IOException
```

There are other methods in this class which we have not indicated. All of these methods can throw an `IOException`. The first `read` method is abstract so the `InputStream` class is abstract. The other two `read` methods can read arrays of bytes or parts of arrays. They are not abstract since they are implemented using the single byte `read` method. The non-abstract subclasses of `InputStream` shown in Figure 11.1 must implement the single byte `read` method. Then any of the three `read` methods declared in the `InputStream` class can be used. Through inheritance this provides a uniform interface for reading a byte stream.

The single byte `read` method does not return the byte read as a value of type `byte` but as a value of type `int` in the range 0 to 255. The reason is that the method is designed to signal end of file by a special return value. If the return type was `byte` there would be no special byte value since all 256 byte values could be present in the stream or file. However, an `int` (32-bits) can hold the byte value as an integer in the range 0 to 255 and a return value of -1 can be used to indicate end of file. All read methods in the byte stream classes use this convention.

■ EXAMPLE 11.11 **(Byte at a time input model)** The following statements show how to construct a `FileInputStream` object associated with a file and process it as a byte stream, one byte at a time:

```
FileInputStream in = new FileInputStream(inFile);
int n = in.read();
while (n != -1)
{
   // process the byte value in n here
   n = in.read(); // try to read another byte
}
```

There is another more compact way that can often be used to write the while-loop:

```
FileInputStream in = new FileInputStream(inFile);
int n;
while ( (n = in.read()) != -1)
{
   // process the byte value in n here
}
```

Here we do both the reading and the end of file testing inside the condition using an assignment statement: the value of `n = in.read()` is `n` and this is compared with the value -1. The extra set of parentheses around the assignment is necessary since `!=` has a higher precedence than assignment. ■

■ EXAMPLE 11.12 **(Buffered byte at a time input model)** For more efficiency we can buffer the input stream and use the statements

```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream(inFile));
int n;
while ( (n = in.read()) != -1)
{
   // process the byte value in n here
}
```

■

■ EXAMPLE 11.13 **(Byte array input model)** Another reading model is to shorten the number of times the body of the while-loop is executed by using the `read` method that reads an array of bytes at a time:

```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream(inFile));

byte[] b = new byte[1024];
int numBytes;
while ( (numBytes = in.read(b)) != -1)
{
   // process b[0] to b[numBytes-1] here
}
```

This version of `read` will try to read a block of 1024 bytes at a time, otherwise it will read less than that number. In particular fewer bytes will be read for the last block unless the file size is an exact multiple of 1024 bytes. In either case, if end of file is not indicated, the method return value indicates how many bytes were actually read and stores them in the specified array `b`. ■

11.6.2 Writing bytes to an output byte stream

The abstract `OutputStream` class declares three methods for writing bytes to an output stream, a method for flushing a stream, and a method for closing a stream. The prototypes are

```
public abstract void write(int b) throws IOException
public void write(byte[] b) throws IOException
public void write(byte[] b, int startPos, int numBytes) throws IOException
public void flush() throws IOException
public void close() throws IOException
```

All of these methods can throw an `IOException`. The first `write` method is abstract so the `OutputStream` class is abstract. It assumes that the byte to be written is the lower 8-bits of an `int` value. The other two `write` methods can write arrays of bytes or parts of arrays. They are not abstract since they are implemented using the single byte `write` method. The non-abstract subclasses of `OutputStream` shown in Figure 11.1 must implement the single byte `write` method. Then any of the three `write` methods declared in the `OutputStream` class can be used. Through inheritance this provides a uniform interface for writing a byte stream.

■ EXAMPLE 11.14 **(Reading and writing bytes)** Here are some statements that open an input and an output file given their names, read the bytes from the input file, process them in some manner, and write the results to the output file:

```
File inFile = new File("in.dat");
File outFile = new File("out.dat");
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream(inFile));
BufferedOutputStream out =
   new BufferedOutputStream(new FileOutputStream(outFile));
int n;
while ( (n = in.read()) != -1)
{
   // process byte value in n here
   out.write(n); // write it to the output file
}
in.close(); // don’t forget to close files
out.close();
```

In the simplest case where no processing is done these statements just copy the input file to the output file like a file copy command. A more efficient version would use the byte array `read` and `write` methods. In this case the loop can be expressed in the form

```
   byte[] b = new byte[1024];
   int numBytes;
   while ( (numBytes = in.read(b)) != -1)
   {
      // process b[0] to b[numBytes-1] here
      out.write(b, 0, numBytes); // write them to the output file
   }
```

Here we have to use the version of `write` that writes out only the first `numBytes` bytes of the array (third argument is the number of bytes to write beginning at index 0 in the array), since `numBytes` may not be 1024 each time, especially when the last buffer is read, unless the file size is a multiple of 1024 bytes. ■

11.6.3 File copy program

Here is a program class called `FileCopier` that uses command line arguments to get the input and output file names and writes a copy of the input file to the output file.

Class `FileCopier`

book-project/chapter11/file_apps

```
package chapter11.file_apps;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * A byte stream (binary) file copy program.
 */
```

```
public class FileCopier
{
   private File inFile;
   private File outFile;

   /**
    * Construct an object for given file objects.
    * @param inFile the input file object
    * @param outFile the output file object
    */
   public FileCopier(File inFile, File outFile)
   {
      this.inFile = inFile;
      this.outFile = outFile;
   }

   /** Perform the byte file copy in blocks of 4096 bytes.
   */
   public void copyFile()
   {
      final int END_OF_FILE = -1;
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      try
      {
         in = new BufferedInputStream(new FileInputStream(inFile));
         out = new BufferedOutputStream(new FileOutputStream(outFile));

         byte[] b = new byte[4096];
         int numBytes;

         while ( (numBytes = in.read(b)) != END_OF_FILE)
         {
            // write bytes  b[0] to b[numBytes-1]
            out.write(b, 0, numBytes);
         }
      }
      catch (FileNotFoundException e)
      {
         System.out.println("Input file does not exist");
      }
      catch (IOException e)
      {
         System.out.println("Unknown IO error");
      }
      finally // always executed
      {
         try
         {
            if (in != null) in.close();
            if (out != null) out.close();
         }
         catch (IOException e)
         {
            System.out.println("Error closing files");
         }
      }
   }

   public static void main(String[] args)
   {
      if (args.length == 2)
      {
         File inFile = new File(args[0]);
         File outFile = new File(args[1]);
         FileCopier copier = new FileCopier(inFile, outFile);
         copier.copyFile();
      }
      else
      {
         System.out.println("args: inFileName outFileName");
      }
   }
}
```

In the `copyFile` method we catch all possible errors using a `try`-`catch`-`finally` block. In this case it is not necessary to include `throws` clauses on any methods. The `finally` block will always be executed, whether or not any exceptions are thrown, so we use it to close any open files.

To try this program from the command line navigate to the `book-project` directory and use the command

```
java chapter11.file_apps.FileCopier files/in.dat files/out.dat
```

where `in.dat` is the name of an existing file, assumed to be in a subdirectory called `files` of the `book-project` directory, and `out.dat` is the name of the output file.

One deficiency in our program is that if the output file already exists then its contents will be lost. A better program would warn the user and ask if the output file should be replaced. The `File` class has methods to do this.

Another way to get file names is with a `JFileChooser` object. Here is a version of `FileCopier` that uses it.

Class `FileCopyChooser`

book-project/chapter11/file_apps

```
package chapter11.file_apps;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.swing.JFileChooser;

/**
 * A byte stream (binary) file copy program.
 */
public class FileCopyChooser
{
   private File inFile;
   private File outFile;

   /**
    * Construct an object for given file objects.
    * @param inFile the input file object
    * @param outFile the output file object
    */
   public FileCopyChooser(File inFile, File outFile)
   {
      this.inFile = inFile;
      this.outFile = outFile;
   }

   /** Perform the byte file copy in blocks of 4096 bytes.
    */
   public void copyFile()
   {
      final int END_OF_FILE = -1;
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      try
      {
         in = new BufferedInputStream(new FileInputStream(inFile));
         out = new BufferedOutputStream(new FileOutputStream(outFile));

         byte[] b = new byte[4096];
         int numBytes;

         while ( (numBytes = in.read(b)) != END_OF_FILE)
         {
            // write bytes  b[0] to b[numBytes-1]
            out.write(b, 0, numBytes);
         }
      }
      catch (FileNotFoundException e)
      {
         System.out.println("Input file does not exist");
      }
      catch (IOException e)
      {
         System.out.println("Unknown IO error");
      }
      finally // always executed
      {
         try
         {
            if (in != null) in.close();
            if (out != null) out.close();
         }
         catch (IOException e)
         {
             System.out.println("Error closing files");
         }
      }
   }

   public static void main(String[] args) throws IOException
   {
      // NOTE: When using FileChooser it is necessary to use
      // file objects everywhere, not file names

      JFileChooser chooser = new JFileChooser();
      int returnValue;

      // Choose an input file object

      returnValue = chooser.showOpenDialog(null);
      if (returnValue == JFileChooser.ERROR_OPTION ||
          returnValue == JFileChooser.CANCEL_OPTION)
      {
         // no input file was chosen
         return;
      }
      File inFile = chooser.getSelectedFile();

      // Choose an output file object

      returnValue = chooser.showSaveDialog(null);
      if (returnValue == JFileChooser.ERROR_OPTION ||
         returnValue == JFileChooser.CANCEL_OPTION)
      {
         // no output file was chosen
         return;
      }
      File outFile = chooser.getSelectedFile();

      FileCopyChooser program = new FileCopyChooser(inFile, outFile);
      program.copyFile();
   }
}
```

Here a `JFileChooser` object is created. When the `showOpenDialog` method is called a dialog box appears and you can navigate to any file and select it. A return code can be used to determine whether the user selected a file or cancelled without choosing a file. The selected input file can be obtained using the `getSelectedFile` method, which returns a `File` object that is used in the `FileInputStream` constructor. Similarly, the `showSaveDialog` method is used to specify an output file in the `FileOutputStream` constructor.

If you want the file name without the full path you can use the `getName` method in the `File` class.

11.7 Reading and writing character streams

A byte stream whose bytes are encoded Unicode characters is called a character stream (also called a text stream). Such character streams are intended to be "human-readable". As mentioned previously they are composed of lines separated by one or more end of line characters. Subclasses of the `Reader` and `Writer` classes are used to process the underlying byte streams as character streams.

When a character stream is opened for reading using a stream constructor, there are various read methods for reading characters sequentially from the stream. If the stream is connected to a file these methods will read characters from the file. Similarly, when a character stream is opened for writing there are various write methods for writing characters sequentially to the stream and if the stream is connected to a file they will write characters to the file.

11.7.1 Reading characters from an input stream

The abstract `Reader` class declares three methods for reading characters from a stream and a method for closing a stream. The prototypes are

```
public abstract int read() throws IOException
public int read(char[] c) throws IOException
public int read(char[] c, int startPos, int numChars) throws IOException
public abstract void close() throws IOException
```

There are other methods in this class which we have not indicated. All of these methods can throw an `IOException`. The first `read` method is abstract so that is why the `Reader` class is abstract. The other two `read` methods can read arrays of characters or parts of arrays. These methods are not abstract since they are implemented using the single character `read` method. The non-abstract subclasses of `Reader` shown in Figure 11.2 must implement the single character `read` method. Then any of the three `read` methods declared in the `Reader` class can be used. Through inheritance this provides a uniform interface for reading a character stream.

The single character `read` method does not return the character read as a value of type `char` (16-bits) but as a value of type `int` in the range 0 to 65535. The reason is that the method is designed to signal end of file by a special return value. If the return type was `char`, there would be no special return value since all 65536 character values could be present in the stream or file. However, an `int` (32-bits) can hold the `char` value as an integer in the range 0 to 65535 and the return value -1 can be used to indicate end of file. All read methods in the character stream classes use this convention.

■ EXAMPLE 11.15 **(Character at a time input model)** The statements

```
FileReader in = new FileReader(inFile);
int n;
while ( (n = in.read()) != -1)
{
   // process the char value in n here
}
```

show how to construct a `FileReader` for a file and process it as a character stream, one character at a time. The underlying byte stream is decoded into to a character stream assuming that the stream was written using a default Unicode encoding scheme. ■

■ EXAMPLE 11.16 **(Buffered character at a time input model)** For more efficiency we can use the statements

```
BufferedReader in =  new BufferedReader(new FileReader(inFile));
int n;
while ( (n = in.read()) != -1)
{
   // process the char value in n here
}
```

to buffer the input stream. ■

■ EXAMPLE 11.17 **(Character array input model)** Another reading model is to shorten the number of times the body of the while-loop is executed by using the `read` method that reads an array of characters at a time:

```
BufferedReader in = new BufferedReader(new FileReader(inFile));
char[] c = new char[1024];
int numChars;
while ( (numChars = in.read(c)) != -1)
{
   // process characters c[0] to c[numChars-1] here
}
```

This version of read will try to read a block of 1024 characters at a time, otherwise it will read fewer than this number. In particular, less characters will be read for the last block unless the file size is an exact multiple of 1024 characters. In either case, if end of file is not indicated, the method return value indicates how many characters were actually read and stores them in the specified array `c`. ■

■ EXAMPLE 11.18 **(Line at a time input model)** The most useful feature of `BufferedReader` is that, in addition to the usual `Reader` class methods, it is the only class that has a `readLine` method with prototype

```
public String readLine() throws IOException
```

that can read a line at a time into a string. This makes sense since character streams are streams of lines. The end of line separator character(s) are not stored in the string. If there are no more lines to read from the stream then the method returns `null` as an end of file indicator. Therefore, for buffered character stream processing we also have the convenient model

```
BufferedReader in = new BufferedReader(new FileReader(inFile));
String line;
while ( (line = in.readLine()) != null)
{
   // process the line here
}
```

for reading a file a line at a time. ■

11.7.2 Writing characters to an output stream

The abstract `Writer` class declares five methods for writing characters to an output stream, a method for flushing a stream, and a method for closing a stream. The prototypes are

```
public void write(int c) throws IOException
public abstract void write(char[] c, int startPos,
   int numChars) throws IOException
public void write(char[] c) throws IOException
public void write(String s, int startPos, int numChars) throws IOException
public void write(String s) throws IOException
public abstract void flush() throws IOException
public abstract void close() throws IOException
```

All of these methods can throw an `IOException`. Here the single character `write` method is not abstract but one of the array writing methods is abstract so the `Writer` class is abstract. The single character `write` method assumes that the character to be written is the lower 16-bits of an `int` value. There are two `write` methods that can write arrays of characters or parts of arrays. Also, there are two similar `write` methods that can write a string or part of a string. The non-abstract subclasses of `Writer` shown in Figure 11.2 must implement the abstract methods so any of the five `write` methods declared in the `Writer` class can be used. Through inheritance this provides a uniform interface for writing a character stream.

■ EXAMPLE 11.19 **(Reading and writing characters)** Here are some statements that read the characters from an input file, process them in some manner and write the results to an output file:

```
File inFile = new File("inFile.dat");
File outFile = new File("outFile.dat");
BufferedReader in = new BufferedReader(new FileReader(inFile));
BufferedWriter out = new BufferedWriter(new FileWriter(outFile));

int n;
while ( (n = in.read()) != -1)
```

```
    {
       // process char value in n here
       out.write(n); // write it to output file
    }
    in.close(); // don't forget to close files
    out.close();
```

A more efficient version would use the `read` and `write` methods that use a `char` array. In this case the loop could be expressed in the form

```
    char[] c = new char[1024];
    int numChars;
    while ( (numChars = in.read(c)) != -1)
    {
       // process c[0] to c[numChars-1] here
       out.write(c, 0, numChars); // write them to output file
    }
```

Here we have to use the version of `write` that writes out only the first `numChars` characters of the array (third argument is the number of characters to write beginning at index 0 in the array), since `numChars` may not be 1024 each time, especially when the last buffer is read, unless the file size is a multiple of 1024 characters. ■

`PrintWriter` class

We are familiar with `System.out`. It is actually a `PrintStream` object that is an anachronism since this class doesn't understand Unicode (early versions of Java did not treat Unicode consistently). It does however have the familiar and very useful `print` and `println` methods which know how to output all the standard data types such as values of `int` and `double` types formatted in "human-readable" form. The `write` methods in the `BufferedWriter` class do not know how to do this. We would like to be able to use `print` and `println` to write data to character streams and files. To do this there is a class called `PrintWriter` in the `Writer` hierarchy that has these methods and also understands Unicode.

■ EXAMPLE 11.20 **(Using `PrintWriter`)** We can use a `PrintWriter` in conjunction with a `BufferedWriter` and a `FileWriter` to write formatted data to a file as the following statements show:

```
    Scanner console = new Scanner(System.in);
    PrintWriter out =
       new PrintWriter(new BufferedWriter(new FileWriter(outFile)));
    System.out.println("What is your name?");
    String name = console.nextLine();
    System.out.println("How old are you?");
    int age = console.nextInt();
    console.nextLine(); // eat end of line
    out.println(name);
    out.println(age);
    out.close();
```

Here `System.out` writes to the console and the `out` object writes to a file. Thus, the person's name and age are written to the file `outFile` using the `out` object. ■

11.7.3 Simple search program

As an example of character stream processing suppose we want to read a character file (text file) a line at a time and write to an output file only those lines that contain a given search pattern.

To do this we can use a `BufferedReader` to read each line of the file as a string using `readLine`, an use the `indexOf` method in the `String` class to search the line for the pattern. If the pattern is found we use a `PrintWriter` to write the line to the output file. Assuming that `in` is the input file reader object and `out` is the output file writer object, the following simple while loop does the line at a time processing and conversion:

```
String line;
while ( (line = in.readLine()) != null)
{
   if (line.indexOf(pattern) >= 0)
      out.println(line);
}
in.close(); // don't forget to close the files
out.close();
```

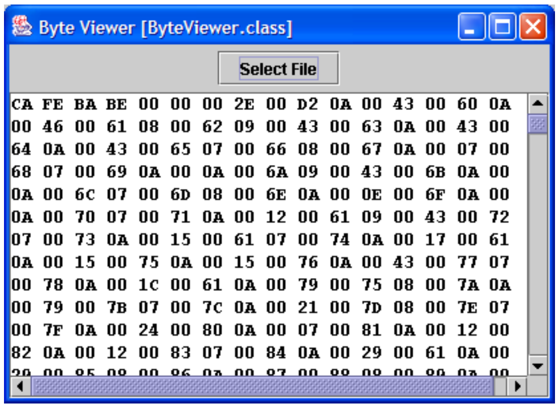
Here is a program class for searching a text file.

Class `FileSearcher`

`book-project/chapter11/file_apps`

```
package chapter11.file_apps;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * A character stream file searcher class.
 * The input file is read a line at a time.
 * Only the lines containing a specified pattern are written to
 * the output file.
 */
public class FileSearcher
{
   File inFile;  // the input file object
   File outFile; // the output file object
   String pattern; // the pattern to search for

   /** Construct a searcher object given file names and pattern.
    * @param inFile the input file object
    * @param outFile the output file object
    * @param pattern the pattern to find in lines of input file
    */
   public FileSearcher(File inFile, File outFile, String pattern)
   {
       this.inFile = inFile;
       this.outFile = outFile;
       this.pattern = pattern;
   }

   /**
    * Perform the file search one line at a time.
    */
   public void searchFile() throws IOException
   {
      BufferedReader in = new BufferedReader(new FileReader(inFile));
      PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(outFile)));
      String line;
      while ( (line = in.readLine()) != null)
      {
         // if pattern is found write line to output file
         if (line.indexOf(pattern) >= 0)
            out.println(line);
      }
      in.close();
      out.close(); // don’t forget this
   }

   public static void main(String[] args) throws IOException
   {
      if (args.length == 3)
      {
         File inFile = new File(args[0]);
         File outFile = new File(args[1]);
         FileSearcher searcher = new FileSearcher(inFile, outFile, args[2]);
         searcher.searchFile();
      }
      else
      {
         System.out.println("args: inFileName outFileName pattern");
      }
   }
}
```

This class shows that we can avoid using any `try` blocks and let the java interpreter catch all exceptions as long as we use the `throws` clause on the `main` method and on the `searchFile` method.



```
CA FE BA BE 00 00 00 2E 00 D2 0A 00 43 00 60 0A
00 46 00 61 08 00 62 09 00 43 00 63 0A 00 43 00
64 0A 00 43 00 65 07 00 66 08 00 67 0A 00 07 00
68 07 00 69 0A 00 0A 00 6A 09 00 43 00 6B 0A 00
0A 00 6C 07 00 6D 08 00 6E 0A 00 0E 00 6F 0A 00
0A 00 70 07 00 71 0A 00 12 00 61 09 00 43 00 72
07 00 73 0A 00 15 00 61 07 00 74 0A 00 17 00 61
0A 00 15 00 75 0A 00 15 00 76 0A 00 43 00 77 07
00 78 0A 00 1C 00 61 0A 00 79 00 75 08 00 7A 0A
00 79 00 7B 07 00 7C 0A 00 21 00 7D 08 00 7E 07
00 7F 0A 00 24 00 80 0A 00 07 00 81 0A 00 12 00
82 0A 00 12 00 83 07 00 84 0A 00 29 00 61 0A 00
```

Figure 11.5: The `ByteViewer` GUI showing `ByteViewer.class`

You can run this class from the command line using

```
java chapter11.files.FileSearcher files/in.dat files/out.dat hello
```

or from `BlueJ` by constructing a `FileSearcher` object and using its `searchFile` method.

11.8 Viewing byte contents of files

A binary file is not in "human-readable" form but we can view any file (binary or text) one byte at a time by writing a program that reads the file a byte at a time and shows the code for each byte as a decimal number in the range 0 to 255 or as a hex code `00` to `FF`. Let us write a program called `ByteViewer` that does this using the hex codes for each byte.

We will use a GUI interface that uses `JFileChooser` to get the input file name. Then we will use a `BufferedInputStream` to read the input file as a byte stream and write the "human-readable" version to a `JTextArea` placed inside a `JScrollPane`. The GUI window is shown in Figure 11.5. Here the hex codes of the bytes in the file are displayed 16 per line. This is done by reading the file a block at a time using a block size of 16 bytes. Assuming that `output` is the name of the `JTextArea`, the file processing loop is given by

```
BufferedInputStream in =
   new BufferedInputStream(new FileInputStream(inFile));
byte[] b = new byte[16];
int numBytes;
output.setText("");
while ( (numBytes = in.read(b) ) != -1)
{
   for (int k = 0; k < numBytes; k++)
   {
      output.append(byteToHex(b[k]) + " ";
   }
   output.append("\n");
```

```
    }
    in.close();
```

The only tricky part is the `byteToHex` method which takes a byte value and converts it to a two character hex string. This is complicated by the fact that a byte is signed (values are treated as in the range -128 to 127 instead of 0 to 255). We first need to convert the byte value to an integer value in order to use the `toHexString` method in the `Integer` class. To convert a `byte` value `b` to an `int` value `i` it is necessary to use the magic statement

```
int i = b & 0x000000FF;
```

This cancels the sign extension which occurs when a signed value in the range -128 to 127 is converted to an integer. Then the integer will contain a value in the range 0 to 255. Therefore the method is given by

```
private String byteToHex(byte b)
{
   int i = b & 0x000000FF;
   String hex = Integer.toHexString(i).toUpperCase();
   if (hex.length() == 1) hex = "0" + hex;
   return hex;
}
```

11.8.1 `ByteViewer` class

Here is the complete class:

Class `ByteViewer`

book-project/chapter11/file_apps

```
package chapter11.file_apps;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
```

```
/**
 * GUI application to display the contents of a file
 * as a sequence of byte values given by their hexadecimal
 * value in the range 00 to FF.
 */
public class ByteViewer extends JFrame
{
   private JTextArea output; // bytes are displayed here
   private JFileChooser chooser;
   private String titleBar = "Byte Viewer";

   /** Construct the GUI */
   public ByteViewer()
   {
      setTitle(titleBar);
      JButton view = new JButton("Select File");
      output = new JTextArea(20,50);
      output.setEditable(false);
      output.setFont(new Font("Courier New", Font.BOLD, 14));
      chooser = new JFileChooser();

      JPanel p = new JPanel();
      p.setLayout(new FlowLayout());
      p.add(view);

      Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      cp.add(p, BorderLayout.NORTH);
      cp.add(new JScrollPane(output), BorderLayout.CENTER);

      view.addActionListener(new ViewButtonHandler());
   }

   private class ViewButtonHandler implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         viewFile();
      }
   }

   private void viewFile()
   {
      // Get the file to view

      int status = chooser.showOpenDialog(null);
      if (status == JFileChooser.ERROR_OPTION ||
          status == JFileChooser.CANCEL_OPTION)
      {
         return;
      }
      File inFile = chooser.getSelectedFile();
      setTitle(titleBar + " ["+ inFile.getName() + "]");

      BufferedInputStream in = null;
      try
      {
         in = new BufferedInputStream(new FileInputStream(inFile));

         // Read file one byte at a time and display the bytes
         // as hexadecimal numbers, 16 per line

         byte[] b = new byte[16];
         int numBytes;
         output.setText("");
         while ((numBytes = in.read(b)) != -1)
         {
            for (int k = 0; k < numBytes; k++)
            {
               output.append(byteToHex(b[k]) + " ");
            }
            output.append("\n");
         }
         in.close();
      }
      catch (IOException e)
      {
         // must catch it because a throws clause cannot be
         // put on the actionPerformed method
         e.printStackTrace();
      }
   }

   /* Convert a byte value, b, to a 2 character hex string
    * of form "HH" where each H is a hex digit 0 to 9, A to F
    */
   private String byteToHex(byte b)
   {
      int i = b & 0x000000FF; // cancel sign-extension
      String hex = Integer.toHexString(i).toUpperCase();
      if (hex.length() == 1) hex = "0" + hex;
      return hex;
   }

   public static void main(String[] args)
   {
      ByteViewer viewer = new ByteViewer();
      viewer.pack(); // instead of setSize
      viewer.setVisible(true);
      viewer.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

You can run this class from the command line or from `BlueJ` by constructing a `ByteViewer` object and using its `view` method.

11.9 Text files of `BankAccount` objects

In this section we consider two ways to represent a database of `BankAccount` objects using a sequential text file. The results can easily be generalized to other kinds of objects. Each object is represented in the file by a record containing a number of fields. Each field corresponds to one of the object's data fields. The following text file formats will be illustrated:

File of multi-line records Each line of the file corresponds to one data field. A typical bank account record would occupy three lines of the file, for example

```
1399527077
Cecil Patterson
22158.11
```

These lines specify the account number, name, and balance of an account.

File of single-line field separated records Each line corresponds to one object (record). The three fields are separated by a special character (delimiter). We cannot use a space as delimiter since owner names may have spaces in them. Therefore we choose the colon character so the above record example would appear in the file as the line

```
1399527077:Cecil Patterson:22158.11
```

For each of these file formats we could design a `readAccount` method to read a `BankAccount` object from a file and a `writeAccount` method to write a `BankAccount` object to a file. We consider here only the single-line colon-separated format. Since the files are text files we will use inheritance and extend the `BufferedReader` and `PrintWriter` classes to do account I/O.

11.9.1 Reading and writing single-line records

To make it easy to read and write bank account objects one at a time it is useful to write methods `readAccount` and `writeAccount` for bank account objects. To develop these methods let us assume prototypes of the form

```
public BankAccount readAccount()
public void writeAccount(BankAccount a)
```

implying that these will be instance methods in some class. Here the `readAccount` method returns a reference to the next `BankAccount` object read from a `BufferedReader`. We use a `BufferedReader` because our file is a text file of lines and this class has a `readLine` method. Similarly, the `writeAccount` method writes an account to a `PrintWriter` since it has the `println` methods.

Implementing the `writeAccount` method

The `writeAccount` method is simple. It just needs to write out the three account fields on one line separated by a colon using the `println` method:

```
public void writeAccount(BankAccount a)
{
   out.println(a.getNumber() + ":" + a.getName() + ":" + a.getBalance());
}
```

where we are assuming that `out` is a `PrintWriter` object.

Implementing the `readAccount` method

At first sight the `readAccount` method seems complicated. We can use `readLine` to read a line as a record but how do we extract the three colon separated fields? There is a class called `StringTokenizer` in the `java.util` package that is designed precisely to solve this problem. The name arises from the fact that a tokenizer breaks its input into pieces called tokens. This class has a constructor of the form

```
public StringTokenizer(String s, String delimiters)
```

and a method of the form

```
public String nextToken()
```

The idea here is that `nextToken()` returns the next field of the string `s` using the characters in `delimiters` as field separators. Therefore, we can read a line of the file with colon-separated fields and construct a `StringTokenizer` object for it using

```
StringTokenizer t = new StringTokenizer(line, ":");
```

Now `t.nextToken()` returns the next field in `line` as a string so our first attempt at `readAccount` is

```
public BankAccount readAccount()
{
   String field1, field2, field3;
   String line = in.readLine();
   if (line == null) return null; // end of file encountered
   StringTokenizer t = new StringTokenizer(line, ":");
   field1 = t.nextToken(); // account number
   field2 = t.nextToken(); // account owner name
   field3 = t.nextToken(); // account balance
   int number = Integer.parseInt(field1.trim());
   double balance = Double.parseDouble(field3.trim());
   return new BankAccount(number, field2.trim(), balance);
}
```

However, we should check for errors. We can use the fact that a NoSuchElementException is thrown by nextToken if there are no more fields in the line to return. This is an unchecked exception. Here is a better version incorporating error checking.

```
public BankAccount readAccount(BufferedReader in) throws IOException,
   EOFException
{
   int number = 0;
   double balance = 0.0;

   String field1, field2, field3;
   String line = in.readLine();

   // Check for normal end of file and return null to indicate eof

   if (line == null) return null;

   StringTokenizer t = new StringTokenizer(line, ":");

   // Check for a partial record

   try
   {
      field1 = t.nextToken();
      field2 = t.nextToken();
      field3 = t.nextToken();
   }
   catch (NoSuchElementException e)
   {
      throw new EOFException("Partial record encountered");
   }

   // check that numeric fields are valid using NumberFormatException
   // (an unchecked exception) and rethrow it if they are not.

   try
   {
      number = Integer.parseInt(field1.trim());
   }
   catch (NumberFormatException e)
   {
      throw new NumberFormatException("Invalid account number");
   }

   try
   {
      balance = Double.parseDouble(field3.trim());
   }
   catch (NumberFormatException e)
   {
      throw new NumberFormatException("Invalid bank balance");
   }

   return new BankAccount(number, field2.trim(), balance);
}
```

11.9.2 Finding a home for `readAccount` and `writeAccount`

An input stream of bank accounts is like a `BufferedReader` with an extra `readAccount` method and an output stream of bank accounts is like a `PrintWriter` that has an extra `writeAccount` method. This suggests that we extend `BufferedReader` to obtain a `BufferedAccountReader` class and that we extend `PrintWriter` to obtain a `PrintAccountWriter` class. Then we can connect files to these streams using `FileReader` and `FileWriter`.

11.9.3 Extending the `BufferedReader` class

The `BufferedAccountReader` subclass of `BufferedReader` has the structure

```
   public class BufferedAccountReader extends BufferedReader
   {
      public BufferedAccountReader(Reader in) {...}
      public BankAccount readAccount()
         throws IOException, EOFException {...}
   }
```

Here we have used one of the same constructor prototypes that are present in the `BufferedReader` class (see Java class documentation, there are other constructors that could also be provided). We accept all methods from `BufferedReader`, in particular the `readLine` method, and we simply add our new `readAccount` method. The constructor can simply make a call to `super` so the complete class is

Class `BufferedAccountReader`

book-project/chapter11/bank_account

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.IOException;
import java.io.Reader;
import java.util.NoSuchElementException;
import java.util.StringTokenizer;

/**
```

```
 *  A class to read <code>BankAccount</code> objects from a buffered
 *  text stream. Each <code>BankAccount</code> object is represented
 *  by one line in the file having a format such as
 *  <pre>
 *     122234422:Frank Johnson:345.56
 *  </pre>
 *  Here the colon character is used to separate the account number
 *  from the name and the balance. The following loop can be used to
 *  process a file of <code>BankAccount</code> objects.
 *  <pre>
 *     BufferedAccountReader in =
 *        new BufferedAccountReader(new FileReader(inFile));
 *     BankAccount a = in.readAccount();
 *     while (a != null)
 *     {
 *        // process account a here
 *        in.readAccount(); // read next account
 *     }
 *  </pre>
 *  In many cases the while loop can be shortened to
 *  <pre>
 *     while ( (a = in.readAccount()) != null)
 *     {
 *        // process account a here
 *     }
 *  </pre>
 */

public class BufferedAccountReader extends BufferedReader
{
   /**
    * Create a buffering <code>BufferedAccountReader</code> input stream
    * that uses a default-sized input buffer.
    * @param in A <code>Reader</code>
    */
   public BufferedAccountReader(Reader in)
   {
      super(in);
   }

   /**
    * Read a <code>BankAccount</code> object.
    * @return the <code>BankAccount</code> read or <code>null</code>
    *   if end of file occurred.
    * @throws <code>EOFException</code> if a partial record is found
    * @throws <code>IOException</code> if an I/O error occurs
    * @throws <code>NumberFormatException</code> if an invalid numeric
    *    field is encountered.
    */
   public BankAccount readAccount() throws IOException, EOFException
   {
      int number = 0;
      double balance = 0.0;

      String field1, field2, field3;
      String line = readLine();

      // Check for normal end of file and return null to indicate eof

      if (line == null) return null;

      StringTokenizer t = new StringTokenizer(line, ":");

      // Check for a partial record

      try
      {
         field1 = t.nextToken();
         field2 = t.nextToken();
         field3 = t.nextToken();
      }
      catch (NoSuchElementException e)
      {
         throw new EOFException("Partial record encountered");
      }


      // check that numeric fields are valid using NumberFormatException
      // (an unchecked exception) and rethrow it if they are not.

      try
      {
         number = Integer.parseInt(field1.trim());
      }
      catch (NumberFormatException e)
      {
         throw new NumberFormatException("Invalid account number");
      }

      try
      {
         balance = Double.parseDouble(field3.trim());
      }
      catch (NumberFormatException e)
      {
         throw new NumberFormatException("Invalid bank balance");
      }

      return new BankAccount(number, field2.trim(), balance);
   }
}
```

Notice that we use `readLine` directly here (`this.readLine()`) since it is inherited from the parent class.

11.9.4 Extending the `PrintWriter` class

The `PrintAccountWriter` subclass of `PrintWriter` has the structure

```
public class PrintAccountWriter extends PrintWriter
{
   public PrintAccountWriter(Writer out) {...}
   public PrintAccountWriter(OutputStream out) {...}
   public void writeAccount(BankAccount a) {...}
}
```

Here we have used only two of the constructor prototypes that are present in the `PrintWriter` class (see Java class documentation). We need the one that takes an `OutputStream` as a parameter since we also want to be able to write to the standard output `System.out` using a `PrintWriter`.

We accept all methods from `PrintWriter`, in particular the `print` and `println` methods, and we simply add our new `writeAccount` method. The constructors can simply make a call to `super` so the complete class is

Class `PrintAccountWriter`

book-project/chapter11/bank_account

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import java.io.OutputStream;
import java.io.Writer;
import java.io.PrintWriter;

/**
 * A class to write <code>BankAccount</code> objects to a
 * <code>PrintWriter</code> text stream. Each <code>BankAccount</code>
 * object is written in the single-line colon-separated format that
 * <code>AccountBufferedReader</code> can read.
 */
public class PrintAccountWriter extends PrintWriter
{
   /**
    * Create a new <code>PrintAccountWriter</code>, without automatic line
    * flushing from an existing <code>Writer</code>.
    * @param out A character output stream
    */
   public PrintAccountWriter(Writer out)
   {
      super(out);
   }

   /**
    * Create a new <code>PrintAccountWriter</code>, without automatic line
    * flushing,  from an existing <code>OutputStream</code>.
    * This convenience constructor creates the necessary intermediate
    * <code>OutputStreamWriter</code>, which will convert characters
    * into bytes using the default character encoding.
    * @param out An output stream
    */
   public PrintAccountWriter(OutputStream out)
   {
      super(out);
   }

   /**
    * Write fields in the single-line colon-separated format understood
    * by <code>BufferedAccountReader</code>.
    * @param a the <code>BankAccount</code>
    */
   public void writeAccount(BankAccount a)
   {
      println(a.getNumber() + ":" + a.getName() + ":" + a.getBalance());
   }
}
```

Notice that we use `println` directly here (`this.println`) since it is inherited from the parent class.

11.10 Bank account text file processing

We can now illustrate how `BankAccount` database files can be processed sequentially using the `BufferedAccountReader` and `PrintAccountWriter` classes. We assume the existence of a small file of 10 `BankAccount` objects called `accounts.txt` containing

```
139952707:Cecil Patterson:22158.11
105870087:Woody Adamson:797.71
201756613:David Irving:941.38
643908962:Gerry Laforge:63606.09
442753562:Mary Lavigne:34434.45
218560951:Mike Tessier:70366.62
969167302:Dave Jenkins:899.24
670350355:Carol Schwartz:5643.45
707959408:Kim Flintstone:15418.01
176306464:Bob Stevenson:5921.03
```

To use `BufferedAccountReader` we can connect it to an account file using the declaration

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFile));
```

Similarly, to use `PrintAccountWriter` we can buffer it and connect it to an account file using the declaration

```
PrintAccountWriter out =
   new PrintAccountWriter(new BufferedWriter(new FileWriter(outFile)));
```

■ EXAMPLE 11.21 **(Reading one record at a time with a while loop)** A model for reading an account file to do some sequential processing is

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFile));
BankAccount a = in.readAccount();
while (a != null)
{
   // process account a here
   a = in.readAccount(); // read next account
}
in.close();
```

Often this can be written in the compact form

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFile));
BankAccount a;
while ((a = in.readAccount()) != null)
{
   // process account a here
}
in.close();
```

Thus, using inheritance and `null` to indicate end of file has given us a very simple processing model. ■

■ EXAMPLE 11.22 **(Displaying a database file in the terminal window)** The simplest kind of processing is to read a `BankAccount` database file and just display its contents in the terminal window (command prompt window). We need to use `System.out` to display data on the console. This object is from the `PrintStream` class which is not a `PrintWriter`. However, it can easily be converted to one using the declaration

```
PrintWriter out = new PrintWriter(System.out);
```

Since `PrintAccountWriter` is a subclass we can use

```
PrintAccountWriter out = new PrintAccountWriter(System.out);
```

Therefore the statements

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFile));
PrintAccountWriter out = new PrintAccountWriter(System.out);
BankAccount a;
while ((a = in.readAccount()) != null)
{
   out.writeAccount(a);
```

```
    }
    in.close();
    out.close();
```

will display a database file in the terminal window. ■

■ EXAMPLE 11.23 **(Copying a database file)** The statements

```
    BufferedAccountReader in =
       new BufferedAccountReader(new FileReader(inFile));
    PrintAccountWriter out =
       new PrintAccountWriter(new BufferedWriter(new FileWriter(outFile)));
    BankAccount a;
    while ((a = in.readAccount()) != null)
    {
       out.writeAccount(a);
    }
    in.close();
    out.close();
```

will make a copy of a database file. ■

In these three examples we didn't need to buffer the input since `BufferedAccountReader`, as a subclass of `BufferedReader`, is buffered.

11.10.1 Finding the maximum balance among the accounts

We now write a class to read an account file and find the maximum balance among all accounts. To do this we can read the first account and assume it has the maximum balance, and then read the remaining accounts and compare balances. The following statements do this kind of processing.

```
    // read first account and assume it has maximum balance
    BankAccount a = in.readAccount();
    double maxBalance = a.getBalance();
    while (a != null)
    {
       if (a.getBalance() > maxBalance)
       {
          maxBalance = a.getBalance();
       }
       a = in.readAccount();
    }
    in.close();
```

Here is the complete program that finds the maximum balance and displays it.

Class `MaxBalanceCalculator`

`book-project/chapter11/bank_account`

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

/**
 * Read bank accounts from a file and display the maximum balance
 */
public class MaxBalanceCalculator
{
   private File inFile;

   /** Construct object for given file object
    * @param inFile the input file object
    */
   public MaxBalanceCalculator(File inFile)
   {
      this.inFile = inFile;
   }

   /** Read the file, calculate maximum balance and display it.
    * @throws java.io.FileNotFoundException
    * @throws java.io.IOException
    */
   public void findMaxBalance() throws FileNotFoundException, IOException
   {
      BufferedAccountReader in = null;
      try
      {
         in = new BufferedAccountReader(new FileReader(inFile));

         BankAccount a = in.readAccount();
         double maxBalance = a.getBalance();

         while (a != null)
         {
            if (a.getBalance() > maxBalance)
            {
               maxBalance = a.getBalance();
            }
            a = in.readAccount();
         }
         in.close();
         System.out.println("Max balance is " + maxBalance);
      }
      finally
      {
         if (in != null) in.close();
      }
   }

   public static void main(String[] args)
      throws FileNotFoundException, IOException
   {
      if (args.length == 1)
      {
         File inFile = new File(args[0]);
         MaxBalanceCalculator calc = new MaxBalanceCalculator(inFile);
         calc.findMaxBalance();
      }
      else
      {
         System.out.println("args: inFileName");
      }
   }
}
```

11.10.2 Account processing

As another bank account processing example consider the problem of reading a file of bank accounts such as `accounts.txt` and writing a new file containing only the accounts whose balances are less than $1000.

■ EXAMPLE 11.24 **(Under 1000 processor)** Assuming that `in` represents the input file and `out` represents the output file, the statements

```
BankAccount a = in.readAccount();
while (a != null)
{
   if (a.getBalance() < 1000.0)
      out.writeAccount(a);
   a = in.readAccount();
}
in.close();
out.close();
```

write the output file. ■

11.10.3 Reading database files into arrays

If a `BankAccount` database file is not too large it may be possible to read it into an array of `BankAccount` objects in memory and process the objects using the array. Statements such as the following could be used as a processing model.

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFile));
```

```
BankAccount[] accounts = new BankAccount[1000];
BankAccount b;
int index = 0;
while ((b = in.readAccount()) != null)
{
   accounts[index] = b; // store account reference in array
   index = index + 1;
}
int numAccounts = index;
in.close();
```

The final value of `index` will be the number of accounts in the file so we can now process the accounts in the array using a for-loop of the form

```
for (int k = 0; k < numAccounts; k++)
{
   // process account[k] here
}
```

The problem here is that, without reading the entire file and counting the accounts, we have no way of knowing how big to make the array. The size problem can be overcome by using one of the Java Collection classes called `ArrayList` that provides dynamic lists whose size automatically expands as needed.

11.10.4 The Dynamic `ArrayList<E>` Class

A better approach is to use a dynamic array: an array whose size can be adjusted at run-time as needed. The array data structure in Java is not dynamic. Once the size has been specified it cannot be changed. However, there is a generic collection class called `ArrayList<E>` in package `java.util` that implements the `List<E>` interface and has a size that is automatically increased as needed. It is generic because it stores objects of element type `E`. Here we are using the generic features of Java 5. In previous versions an `ArrayList` always stored objects of type `Object`. This type can still be used in Java 5 but results in an unsafe type warning. We will use only following methods (there are many others).

- **`public ArrayList()`**

  Constructor for an **`ArrayList`** object with elements of type **`E`** that has space initially for 10 elements.

- **`public ArrayList(int initialCapacity)`**

  Constructor for an **`ArrayList`** object with elements of type **`E`** that has space initially for **`initialCapacity`** elements.

- **`public int size()`**

  Return the current number of elements in **`this`** list.

- **`public void add(E element)`**

  Add a new **`element`** of type **`E`** at the end of **`this`** list.

- **`public E get(int i)`**

  Return the element of type **`E`** at position **`i`** in **`this`** list (positions begin at 0).

- **`public void set(int i, E element)`**

  Replace the element at position **`i`** of **`this`** list by the given **`element`**.

The syntax is not as convenient as the square bracket notation used for indexing arrays but the dynamic properties of the list are more important here.

■ EXAMPLE 11.25 **(Reading accounts into an array list)** The following statements show how to read an account file into a list object of type `ArrayList<BankAccount>`:

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFileName));
List<BankAccount> accounts = new ArrayList<BankAccount>(100);
BankAccount b;
while ((b = in.readAccount()) != null)
{
   accounts.add(b);
}
```

Here the array list starts out with space for 100 account references and the `add` method adds a new entry at the end of the list. If more than 100 accounts are read the list is automatically expanded to hold the additional accounts. ■

■ EXAMPLE 11.26 **(Processing an array list)** The following statements show how to sequentially process the accounts that were read into a list and add $100 to each balance.

```
List<BankAccount> accounts = new ArrayList<BankAccount>(100);
...
for (int k = 0; k < accounts.size(); k++)
{
   BankAccount b = accounts.get(k);
   b.deposit(100);
   accounts.set(k, b);
}
```

Here the `get` method is used to return a reference to an account given its index in the list and the `set` method is used to update the object at each index. ■

Of course, if we can process all accounts in the database file in one pass, as we read the file, then there is no need to store the file in an array or a list. However, if it is necessary to examine the accounts all at once, or more than once, such as in a sorting algorithm or a binary search algorithm, then the list approach would be useful. This would be the case if you wanted to sort a database file and it fits in an array or a list in memory.

Here is a simple class that illustrates how to store accounts in a list, sort them by name using the bubble sort algorithm (see Chapter 8, Page 404), and display the sorted list.

Class `ListProcessor`

`book-project/chapter11/bank_account`

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * Read bank accounts from a file into an ArrayList and process them.
 * In this example the accounts are simply written to the terminal
 * window from the list. Since lists are dynamic we don’t need to
 * specify the maximum number of accounts as we did using an array.
 */
public class ListProcessor
{
   private File inFile;

   /** Define a list processor for a file.
    * @param inFile the input file object
    */
   public ListProcessor(File inFile)
   {
      this.inFile = inFile;
   }

   /** Read the list, sort it and write it to terminal
    *  @throws FileNotFoundException
    *  @throws IOException
    */
   public void processList()
      throws FileNotFoundException, IOException
   {  // An ArrayList ’is a’ list
      List<BankAccount> accounts = new ArrayList<BankAccount>();
      BufferedAccountReader in = null;
      PrintAccountWriter out = null;

      try
      {
         in =  new BufferedAccountReader(new FileReader(inFile));

         BankAccount b;
         while ( (b = in.readAccount()) != null)
         {
            accounts.add(b); // add reference to end of list
         }

         // Now sort the list

         bubbleSort(accounts);

         // Now write the list to standard output

         out = new PrintAccountWriter(System.out);
         for (int k = 0 ; k < accounts.size(); k++)
         {
            out.writeAccount(accounts.get(k));
         }
      }
      finally
      {
         if (in != null) in.close();
         if (out != null) out.close();
      }
   }

   /**
    * Sort an array list of bank accounts in increasing order by name.
    * @param a the list to sort
    */
   private void bubbleSort(List<BankAccount> a)
   {
      int n = a.size();
      for (int p = 1; p <= n - 1; p++) // loop over passes
      {
         for (int j = 0; j <= n - 1 - p; j++)
         {
            String name1 = a.get(j).getName();
            String name2 = a.get(j+1).getName();
            if (name1.compareTo(name2) > 0)
            {
               BankAccount temp = a.get(j);
               a.set(j, a.get(j+1));
               a.set(j + 1, temp);
            }
         }
      }
   }

   public static void main(String[] args)
      throws FileNotFoundException, IOException
   {
      if (args.length == 1)
      {
         File inFile = new File(args[0]);
         ListProcessor processor = new ListProcessor(inFile);
         processor.processList();
      }
      else
      {
         System.out.println("args: inFileName");
      }
   }
}
```

For the sample text file `accounts.txt` the output is

```
176306464:Bob Stevenson:5921.03
670350355:Carol Schwartz:5643.45
139952707:Cecil Patterson:22158.11
969167302:Dave Jenkins:899.24
201756613:David Irving:941.38
643908962:Gerry Laforge:63606.09
707959408:Kim Flintstone:15418.01
442753562:Mary Lavigne:34434.45
218560951:Mike Tessier:70366.62
105870087:Woody Adamson:797.71
```

which shows that the accounts are sorted by name.

■ EXAMPLE 11.27 **(Using primitive types with `ArrayList`)** The elements that can be stored in an `ArrayList` object must be objects of some type `E`. The primitive types such as `int` or `double` are not objects so they cannot be directly used. However every primitive type has a corresponding object type. For example, `int` and `double` have the wrapper classes `Integer` and `Double`.

To make a list of integers we can use statements such as

```
List<Integer> numbers = new ArrayList<Integer>();
numbers.add(new Integer(1)); \\ list is [1]
numbers.add(new Integer(2)); \\ list is [1,2]
int k = (numbers.get(1)).intValue(); \\ rturn value 2
numbers.set(1, new Integer(3)); \\ list is [1,3]
```

Converting back and forth between the `int` and `Integer` types is clumsy so the concepts of auto-boxing and unboxing were introduced in Java 5 so the above statements can be expressed as

```
List<Integer> numbers = new ArrayList<Integer>();
numbers.add(1); \\ list is [1]
numbers.add(2); \\ list is [1,2]
int k = numbers.get(1); \\ return value 2
numbers.set(1, 3); \\ list is [1,3]
```

For example, in the `add` method `1` is automatically converted to `new Integer(1)` by the compiler (auto boxing) and in the `get` method the `intValue` method is automatically applied to convert the `Integer` object back to the `int` type (auto unboxing). Note however, that it is necessary to use `List<Integer>` and not `List<int>`. ■

11.11 Binary object files using polymorphism

In Chapter 9 we made a `JointBankAccount` class which was a subclass of `BankAccount`. Suppose we now want to create files containing both kinds of accounts. We could do this by modifying our text database structure so that the kind of account was indicated by a code. For example, the single-line format could be modified to have an extra numeric field at the beginning: a value of 1 could represent a single owner account and a value of 2 could represent a joint owner account. Two sample records might be

```
1:121233123:Fred Flintstone:134.56
2:332423232:Barney Rubble:Betty Rubble:3434.45
```

Then we would need to modify the `readAccount` method in the `BufferedAccountReader` class to read the first field and use an if-statement to determine whether to read three more fields and construct a `BankAccount` object or to read four more fields and construct a `JointBankAccount` object. Then the `readAccount` method would return a `BankAccount` reference in either case. An if-statement would also be needed in `writeAccount`. If we added new types of accounts `BufferedAccountReader` and `PrintAccountWriter` would have to be modified again.

There is a better way in Java called **object serialization** that keeps track of all the objects in any hierarchy. To serialize an object means to write out all its data fields in some binary format. If some of the fields are objects then their fields are also serialized and written out, and so on. This is a powerful polymorphic way to write objects to a file and there is an `ObjectOutputStream` class that will do this. For each kind of object you want to write it is only necessary to specify that its class implements the `Serializable` interface. For the `BankAccount` class this is done with the modified class declaration

```
public class BankAccount implements java.io.Serializable
{
  // everything else here is same as before
}
```

The `Serializable` interface has no methods to implement. It simply serves as a tag which says "you have the right to serialize my objects". These rights are automatically granted to all subclasses such as `JointBankAccount`.

11.11.1 Writing serialized objects to a file

To write a `BankAccount` or `JointBankAccount` object in serialized form to a file is very easy. It is only necessary to construct an `ObjectOutputStream`, connect it to the file using a declaration such as

```
ObjectOutputStream out =
   new ObjectOutputStream(new FileOutputStream(outFile));
```

and use its `writeObject` method, which has the prototype

```
public void writeObject(Object obj) throws IOException
```

to write an object to the file. Any kind of serializable object can be written since the argument is of type `Object`.

For our bank account example the easiest way to write some bank account objects to a binary object file is to first put them in a list and then write the list to the file.

■ EXAMPLE 11.28 **(Writing an `ArrayList` object to a file)** Assuming that `accounts` is a list of type `ArrayList<BankAccount>` the statements

```
ObjectOutputStream out =
   new ObjectOutputStream(new FileOutputStream(outFile));
out.writeObject(accounts);
out.close();
```

write it to a binary object file. ■

Here is a short class that writes a mixture of bank account and joint bank account objects to a file after putting them into a list.

Class `AccountListObjectWriter`

`book-project/chapter11/bank_account`

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

/**
 * Store some BankAccount and JointBankAccount objects in an ArrayList
 * and write the list object to file as a serialized object
 */
public class AccountListObjectWriter
{
   private File outFile;
   private List<BankAccount> accounts;

   /** Define an object for some bank accounts.
    * @param outFile the name of the output file object
    */
   public AccountListObjectWriter(File outFile)
   {
      this.outFile = outFile;
      accounts = new ArrayList<BankAccount>();

      accounts.add(new BankAccount(1234, "Cecil Patterson", 22158.11));
      accounts.add(new BankAccount(3215, "Woody, Adamson", 797.71));
      accounts.add(new JointBankAccount(5424, "Wilma Flintstone",
         "Fred Flintstone", 4524.56));
      accounts.add(new BankAccount(6632, "David Irving", 941.38));
      accounts.add(new BankAccount(7614, "Gerry Laforge", 63606.09));
      accounts.add(new BankAccount(5337, "Mary Lavigne", 34434.45));
      accounts.add(new JointBankAccount(5423, "Barney Rubble",
         "Betty Rubble", 4524.56));
      accounts.add(new BankAccount(6206, "Mike Tessier", 70366.62));
      accounts.add(new BankAccount(5210, "Dave Jenkins", 899.24));
      accounts.add(new BankAccount(5313, "Carol Schwartz", 5643.45));
      accounts.add(new BankAccount(7079, "Kim Flintstone", 15418.01));
      accounts.add(new BankAccount(1763, "Bob Stevenson", 5921.03));
      accounts.add(new JointBankAccount(5422, "Carol Schwartz",
         "Dave Jenkins", 47894.23));
   }

   /** Process the list and write list object to output file
    *  @throws IOException
    */
   public void writeAccountList() throws IOException
   {
      ObjectOutputStream out = null;
      try
      {
         // Write the list to a file as an object

         out = new ObjectOutputStream(new FileOutputStream(outFile));
         out.writeObject(accounts);
      }
      finally
      {
         if (out != null) out.close();
      }
   }

   public static void main(String[] args) throws IOException
   {
      if (args.length == 1)
      {
         File outFile = new File(args[0]);
         AccountListObjectWriter writer = new AccountListObjectWriter(outFile);
         writer.writeAccountList();
      }
      else
      {
         System.out.println("args: outFileName");
      }
   }
}
```

Run this program to produce a file called `accounts.obj`. You won't be able to view this file with a text editor.

11.11.2 Reading serialized objects from a file

To read `BankAccount` or `JointBankAccount` objects from an object file such as `accounts.obj` is also very easy. This is called **object deserialization**. Simply construct an `ObjectInputStream`, connect it to the file using a declaration such as

```
ObjectInputStream in =
   new ObjectInputStream(new InputFileStream(inFile));
```

and use its `readObject` method, which has the prototype

```
public Object readObject() throws ClassNotFoundException, IOException
```

to read objects from the file. Since any kind of object can be stored in the file, `readObject` returns an `Object` type so it will be necessary to perform a typecast to recover the actual type of object. If you typecast to the wrong kind of object then a `ClassNotFoundException` is thrown.

■ EXAMPLE 11.29 **(Reading an `ArrayList` object from a binary object file)** The statements

```
ObjectInputStream in =
   new ObjectInputStream(new InputFileStream(inFile));
List<BankAccount> accounts = (List) in.readObject();
in.close();
```

show how easy it is to read the list object created by `AccountListObjectWriter`. ■

Here is a class that reads the list from the binary object file, displays the accounts and computes the total balance of all accounts. The system knows which accounts are bank account objects and which are joint bank account objects.

Class `AccountListObjectReader`

book-project/chapter11/bank_account

```
package chapter11.bank_account;
import custom_classes.BankAccount;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.List;

/**
 * Read account list object that was written by WriteAccountListObject
 * and display accounts in terminal window.
 */
public class AccountListObjectReader
{
   private File inFile;
   /** Construct an object for a given file object.
    * @param inFile the binary input file object
    */
   public AccountListObjectReader(File inFile)
   {
      this.inFile = inFile;

   }

   /** Read the array list and display them in the terminal window
    * and compute the total balance of all accounts.
    * @throws ClassNotFoundException
    * @throws FileNotFoundException
    * @throws IOException
    */
   public void processAccountList()
      throws ClassNotFoundException, FileNotFoundException, IOException
   {
      ObjectInputStream in = null;
      try
      {
         // Read the array list of BankAccount objects and
         // display them on standard output. Also compute the
         // total balance of all accounts and display it

         in = new ObjectInputStream(new FileInputStream(inFile));

         List<BankAccount> accounts = (List) in.readObject();
         in.close();

         double totalBalance = 0.0;
         for (int k = 0; k < accounts.size(); k++)
         {
            BankAccount b = accounts.get(k);
            totalBalance = totalBalance + b.getBalance();
            System.out.println(b);
         }
         System.out.println("Total balance is " + totalBalance);
      }
      finally
      {
         if (in != null) in.close();
      }
   }

   public static void main(String[] args)
      throws ClassNotFoundException, FileNotFoundException, IOException
   {
      if (args.length == 1)
      {
         File inFile = new File(args[0]);
         AccountListObjectReader reader = new AccountListObjectReader(inFile);
         reader.processAccountList();
      }
      else
      {
         System.out.println("args: inFileName");
      }
   }
}
```

The output of the program is

```
BankAccount[1234, Cecil Patterson, 22158.11]
BankAccount[3215, Woody, Adamson, 797.71]
JointBankAccount[BankAccount[5424, Wilma Flintstone, 4524.56], Fred Flintstone]
BankAccount[6632, David Irving, 941.38]
BankAccount[7614, Gerry Laforge, 63606.09]
BankAccount[5337, Mary Lavigne, 34434.45]
JointBankAccount[BankAccount[5423, Barney Rubble, 4524.56], Betty Rubble]
BankAccount[6206, Mike Tessier, 70366.62]
BankAccount[5210, Dave Jenkins, 899.24]
BankAccount[5313, Carol Schwartz, 5643.45]
BankAccount[7079, Kim Flintstone, 15418.01]
BankAccount[1763, Bob Stevenson, 5921.03]
JointBankAccount[BankAccount[5422, Carol Schwartz, 47894.23], Dave Jenkins]
Total balance is 277129.44
```

Reading more than one object at a time

Instead of putting the account objects into a list we could have written them directly to the binary object file one at at time using a loop. Then the binary object file will contain more than one object and it is necessary to read the file using a loop.

The `readObject` method does not use `null` to indicate end of file since a `null` value can appear in a binary object file. Instead, it throws an `EOFException` which is a subclass of `IOException`. Therefore a typical processing loop to read an entire object file and check exceptions has the following structure

```
try
{
   while (true)
   {
      // use in.readObject() here to return next object
   }
}
catch (EOFException e)
{
   // nothing to do here, normal end of file
}
catch (ClassNotFoundException e)
```

```
{
   // display message here
}
finally
{
   if (in != null) in.close(); // may throw IOException
}
```

The while-loop is not really infinite: when the `EOFException` is thrown we jump out of it to the first `catch` block. There is nothing to do in this case. We could close the file but this is done in the `finally` block. The `ClassNotFoundException` is a checked exception so if you don't want to catch it you can put it in the `throws` clause of the containing method.

Here are some statements that read a file of `BankAccount` objects using a while-loop, and display them in readable form in the terminal window with the polymorphic `toString` method used by `println`.

```
try
{
   while (true) // not really an infinite loop
   {
      BankAccount b = (BankAccount) in.readObject();
      System.out.println(b);
   }
}
catch (EOFException e) // normal end of file comes here
{
   // nothing to do, this is the normal exit
}
catch (ClassNotFoundException e)
{
   // indicate error
}
finally // guaranteed to execute before control leaves the method
{
   if (in != null) in.close(); // can throw IOException
}
```

It is important to note that if new subclasses are added to the hierarchy these statements do not need to be changed in order to process objects of the new subclasses: the while-loop is polymorphic. Another important feature of serialization and deserialization is that we do not have to worry about designing a database file format.

11.12 Review exercises

► **Review Exercise 11.1** Define the following terms and give examples of each.

file	stream	input stream
output stream	buffering	sequential file
random access file	reading a sequential file	writing a sequential file
text stream or file	binary stream or file	byte stream or file
character stream or file	Unicode	`InputStream` hierarchy
`OutputStream` hierarchy	`Reader` hierarchy	`Writer` hierarchy
`FileInputStream`	`FileOutputStream`	`BufferedInputStream`
`BufferedOutputStream`	`FileReader`	`InputStreamReader`
`BufferedReader`	`OutputStreamWriter`	`FileWriter`
unchecked exception	`throws` clause	checked exception
`IOException`	`EOFException`	`FileNotFoundException`
`PrintWriter`	object serialization	object deserialization
`ObjectInputStream`	`ObjectOutputStream`	

► **Review Exercise 11.2** Write statements that use a while-loop to read a file a byte at a time and write the bytes to another file.

► **Review Exercise 11.3** Write statements that use a while-loop to read a file using blocks of 4096 bytes at a time and write the bytes to another file.

► **Review Exercise 11.4** Write statements that use a while-loop to read a file a character at a time and write the characters to another file.

► **Review Exercise 11.5** Write statements that use a while-loop to read a file using blocks of 4096 characters at a time and write the characters to another file.

► **Review Exercise 11.6** Write statements that use a while-loop to read a character file a line at a time and write the lines to another file.

► **Review Exercise 11.7** Write statements that use object serialization and a for-loop to write an array of `BankAccount` or `JointBankAccount` objects to a file.

► **Review Exercise 11.8** Write statements that use object deserialization and a while-loop to read an an object file of `BankAccount` or `JointBankAccount` objects.

11.13 Exercises

► **Exercise 11.1 (Line ending characters using `ByteViewer`)**
Use `ByteViewer` to verify the lines of a text file in Windows end in carriage return and line feed (`0D 0A` hex, 13, 10 decimal) and line in Unix end with linefeed only.

► **Exercise 11.2 (Dos to Unix text file conversion)**
Using the `BufferedInputStream` and `FileInputStream` classes, write a class called `DosToUnix` that converts a Dos text file to a Unix text file. Hint: Read the file one byte at a time and write all characters to the output file except carriage return characters.

► Exercise 11.3 (Unix to Dos text file conversion)
Using the `BufferedInputStream` and `FileInputStream` classes, write a class called `UnixToDos` that converts a Unix text file to a Dos text file. Hint: Each time a line feed is encountered write a carriage return and a line feed character to the output file.

► Exercise 11.4 (Changing the separator character)
Write a program class called `ChangeSeparator` that converts an account file in the single-line text format to a new file using a different separator character. Use command line arguments for the separator character, input file name, and output file name.

► Exercise 11.5 (account statistics)
Write a program class called `AccountStatistics` that reads a `BankAccount` database file in the single-line text format, computes the minimum balance, the maximum balance, the average balance, the total amount in all accounts, and displays the results. All processing should be done with a single while-loop. Test your program using `accounts.txt`.

► Exercise 11.6 (account statistics)
Same as previous exercise but use the serialized object format which allows for both `BankAccount` and `JointBankAccount` objects in the file. Use the binary object file called `accounts.obj` created by `AccountListObjectWriter` to test your class.

► Exercise 11.7 (accounts with balances below a threshold)
Write a class called `Under1000Processor` that uses the serialized object format which allows for both bank account and joint bank account objects in the file. The class reads the file `accounts.obj` and produces an output file in binary object format that contains only the accounts that have a balance less tham 1000.

► Exercise 11.8 (A phone book database)
Let us design a file format for a phone book. Each entry for a person is defined by the last name, first name, address, and phone number. Here is a file called `phonebook.txt` containing ten records in the single-line colon-separated format:

```
Garson:Dan:325 Albert St:524-8266
Lafreniere:John:433 Colborne St:522-1114
Roberts:Clarice:123 Grand River St:524-4531
Best:Ringo:250 Paris St:524-9137
Robbins:Chris:29 North Park Rd:523-9493
James:Clifford:72 North Park Rd:523-4394
McDonald:Ed:95 Regent St:524-6383
Damon:Andy:170 Abbey Road:522-9228
Perez:Louise:212 Ramsey Lake Rd:524-2220
Jones:Kim:63 York St:524-9971
```

We can write a class called `PhoneBookEntry` whose objects represent each entry:

```
public class PhoneBookEntry implements java.io.Serializable
{
   private String lastName;
   private String firstName;
   private String address;
   private String phoneNumber;

   public PhoneBookEntry(String last, String first, String addr,
      String number)
   {
      lastName = last;
      firstName = first;
      address = addr;
      phoneNumber = number;
   }

   public String toString()
   {
      return lastName + ":" + firstName + ":" +
         address + ":" + phoneNumber;
   }

   // Enquiry methods to retrieve data fields

   public String getLastName() { return lastName; }
   public String getFirstName() { return firstName; }
   public String getAddress() { return address; }
   public String getPhoneNumber() { return phoneNumber; }
}
```

We make the class serializable in case we also want to use it with the binary object format supported by `ObjectInputStream` and `ObjectOutputStream`.

(a) As in the case of the bank account example, write a `BufferedPhoneBookReader` class that is similar to `BufferedAccountReader` except that instead of a `readAccount` method it has a `readPhoneBookEntry` method to read an entry from a stream.

(b) Similarly, write a `PrintPhoneBookWriter` class that is similar to `PrintAccountWriter` except that instead of a `writeAccount` method it has a `writePhoneBookEntry` method to write an entry to a stream.

(c) To test your classes write a simple copy program called `PhoneBookCopy` which makes a copy of a phone book database file.

► **Exercise 11.9 (Binary database format)**
It is possible to write the standard types such as `int` and `double` to a file in binary rather than text format using the `DataInputStream` and `DataOutputStream` classes, which implement the `DataInput` and `DataOutput` binary I/O interfaces.

(a) Find out how these classes work and extend them to classes `AccountInputStream` and `AccountOutputStream` that can read and write binary files of `BankAccount` objects.

(b) Write a conversion program called `ConvertToBinary` that reads an account database in single-line text format and writes it in the binary format defined by your new classes. You can use a loop of the form

```
BufferedAccountReader in =
   new BufferedAccountReader(new FileReader(inFileName));
AccountOutputStream out = new AccountOutputStream(
   new BufferedOutputStream(new FileOutputStream(outFileName)));
BankAccount a;
while ((a = in.readAccount()) != null)
{
   out.writeAccount(a);
}
in.close();
out.close();
```

Test your program using `accounts.txt` to create a binary version called `accounts.bin`.

(c) Write a program called `MaxBalanceBinary` that, like `MaxBalance`, reads `accounts.bin` and displays the maximum balance.

(d) Use the `ByteViewer` program to have a look at `accounts.bin`. Can you discover how integers are stored in binary format? Hint: what is the connection between the first account number `1399527077` and the first four bytes values (decimal) `83`, `107`, `22`, and `165` in the file

► Exercise 11.10 (Searching for patterns)
Consider the following problem: "Search a text file for the lines containing a given pattern (string) and display only the lines containing the pattern". To do this write a program class called `Search` that takes the pattern string and the file name as command line arguments. For example, running `Search` on the file `accounts.txt` using `Dav` as the pattern should produce

```
java Search Dav accounts.txt
2017566133:David Irving:3941.38
969167302 :Dave Jenkins:1199.24
```

Use the following `BufferedSearchReader` class that extends the `BufferedReader` class and overrides the `readLine` method so that it returns only lines containing the pattern:

```
import java.io.IOException;
import java.io.Reader;
import java.io.BufferedReader;

public class BufferedSearchReader extends BufferedReader
{
   String pattern;
```

```
      public BufferedSearchReader(Reader in, String pattern)
      {
         super(in);
         this.pattern = pattern;
      }

      public String readLine() throws IOException
      {
         String line = super.readLine();
         boolean done = false;

         // skip any lines that don't contain the pattern

         while ( (line != null) && ! done)
         {
            if (line.indexOf(pattern) != -1)
               done = true;
            else
               line = super.readLine();
         }
         return line;
      }
   }
```

► **Exercise 11.11 (A simple student marks database)**
Consider the following file format for student marks in a course:

```
   Johnson, Frank:0134567:56:65:33:67:45
   Nicholson, Al :0023458:67:87:56:76:90
```

Here there are six fields per record. The format is last name, comma, and first name for the first field, followed by the student number, followed by a list of 5 marks out of 100 percent.

(a) Write a class called `Student` that represents this structure. Each `Student` object has a name, student number, and an array of 5 integer marks.

(b) Write a `MarksReader` class to read files with this format. Like `BufferedAccountReader` it should provide a method called `readStudent` for reading one student record.

(c) Write a program called `ClassAverages` that reads a student marks file and writes two output files. The first output file has the following format without the student names.

```
   Student Number   Term Marks           Final Term Mark
   0134567          56, 65, 33, 67, 45                53
   0023458          67, 87, 56 ,78, 90                76
```

The averages are computed using floating point division with rounding to the nearest integer. This file can be printed and posted so that students can check their marks. Hint: Just use a `PrintWriter` connected to a file to do the output. You will find the `Format` class useful for formatting the output in columns

The second output file has the following format

```
Student Name       Student Number   Term Marks            Final Term Mark
Johnson, Frank     0134567          56, 65, 33, 67, 45                 53
Nicholson, Al      0023458          67, 87, 56 ,78, 90                 76
```

that differs from the preceding one only by including an extra column containing the student names in the format last name, comma, space, and first name. The teacher can keep this version for recording the marks.

► Exercise 11.12 (Transaction file processing)
Suppose we have a `BankAccount` database file in the single-line colon-separated format (called the master file) and another file called the transaction file that contains lines of the form

```
126534:D:37.50
452324:W:450.00
...
```

Each line represents a transaction. The first field is the account number, the second field is a transaction code (`D` for deposit and `W` for withdrawal), and the third field is the amount that is withdrawn or deposited.

(a) Write a class called `Transaction` that uses three data fields to encapsulate a transaction.

(b) Write a class called `TransactionReader` that extends `BufferedReader` by providing a `readTransaction` method with prototype

```
        public Transaction readTransaction()
```

that reads one line from a transaction file and returns it as a `Transaction` object.

(c) Assuming that the transaction file and the master account file are each sorted in order of increasing account number, write a program class called `UpdateMasterAccounts` that reads a transaction file and a master file, and uses the transaction file to produce a new updated master file. A pseudo-code algorithm is given in Figure 11.6.

```
ALGORITHM UpdateMasterFile(trans,master,newMaster)
INPUT transaction file trans, master file master
INPUT new master file newMaster
Read record from trans
Read record from master
WHILE not end of file on trans DO
    WHILE master account number < trans account number DO
        Write master record to newMaster file
        Read next record from master file
    END WHILE
    Use trans record to update master record
    Write updated master record to newMaster file
    Read next record from trans file
    Read next record from master file
END WHILE
WHILE not end of file on master DO
    Write master record to newMaster file
    Read next record from master file
END WHILE
```

Figure 11.6: Pseudo-code transaction processing algorithm

Chapter 12

Searching and Sorting Algorithms

With an Introduction to Algorithm Efficiency

Outline

Minimum and maximum algorithms

Linear and binary search algorithm

Running times for linear and binary search

Selection and insertion sort

Running time for selection and insertion sort

Simulation to compare selection and insertion sort

Mergesort

File merge example

Quicksort

Running time for mergesort and quicksort

Simulation to compare mergesort and quicksort

Generic sorting

Sorting strings in lexicographical order

Comparing bank account objects

12.1 Introduction

In this chapter we study some important searching and sorting algorithms with an emphasis on efficiency and recursion. Searching and sorting are two of the most important processing operations performed by computers so it is important to have efficient algorithms. Some of the material from Chapter 8, Section 8.4, is repeated here in a more general form.

First we consider some simple variations of algorithms for finding the minimum or maximum values in an array of n elements and for doing a linear search of an array for a given value. To measure the efficiency of an algorithm we introduce a special mathematical order notation that can be used to measure the "running time" without worrying about the effects of particular computer hardware and software. Using this notation we see that these simple algorithms are $O(n)$ which means that for large n an upper bound on their running time is proportional to n, the number of elements to search. For each algorithm we can have three kinds of behavior: best case, average case, and worst case.

Next we consider both recursive and non-recursive versions of the binary search algorithm for a sorted array and determine the running time. It will be clear that binary search is much more efficient for large arrays than linear search since we can show it is $O(\log n)$, whereas linear search is $O(n)$.

Next we consider the efficiency of four popular sorting algorithms. Two of these, selection sort and insertion sort, are called quadratic sorting algorithms because they are $O(n^2)$. The other two, mergesort and quicksort are $O(n \log n)$. This means that they are much more efficient for large arrays than the quadratic algorithms. The mergesort and quicksort algorithms are naturally recursive. We also show how to empirically compare sorting algorithms using a simulation.

Initially our algorithms are developed using arrays of type `int[]` but there are many other kinds of arrays that we might want to search or sort. Rather than rewrite each algorithm for different types of arrays we develop generic searching and sorting classes that work with arrays of type `Object[]` using the `Comparable` and `Comparator` interfaces to define an ordering for the elements.

12.2 Minimum and maximum algorithms

We have considered these algorithms in Chapter 8, Section 8.4, and now want to generalize them so that a specified subarray, rather than the entire array, is searched. A **subarray** of an array $\langle a_0, \ldots, a_{n-1} \rangle$ is defined to be any sequence of elements $\langle a_{start}, \ldots, a_{end} \rangle$ with $0 \le start \le end \le n-1$. The entire array is the subarray having $start = 0$ and $end = n-1$.

The minimum problem can be stated formally as follows:

> "Given the array $\langle a_0, \ldots, a_{n-1} \rangle$ and the index values $start$ and end defining a subarray, determine an index i such that $start \le i \le end$ and $a_i \le a_k$ for all k such that $start \le k \le end$."

The index i is not unique since the minimum value may occur at several places in the subarray. Even for a simple algorithm like this there are several variations. Do we return the minimum value or the position at which the minimum value occurs? If we return the index do we return the index for the first occurrence of the minimum or the last occurrence?

```
**ALGORITHM** FindMinimum($\langle a_0, a_1, \ldots, a_{n-1}\rangle$, $start$, $end$)
$index \leftarrow start$
**FOR** $k \leftarrow start + 1$ **TO** $end$ **DO**
    **IF** $a_k < a_{index}$ **THEN**
        $index \leftarrow k$
    **END IF**
**END FOR**
**RETURN** $index$
```

Figure 12.1: Pseudo-code algorithm for minimum array element

Let us first design the algorithm to return the position of the first occurrence of the minimum. We begin by assuming the minimum value is at index *start* and then use a loop to process the remaining elements in the subarray from $start + 1$ to end. Each time a smaller value is obtained the index is updated. When the loop terminates the index will be the position of the first occurrence of the minimum. The pseudo-code algorithm is given in Figure 12.1. The final value of *index* is the smallest index of the subarray such that a_{index} is the minimum value of the elements in the subarray.

Here are three variations of `findMinimum` in Java.

(a) Return index of first occurrence:

```
int findMinimum(int[] a, int start, int end)
{
   int index = start;
   for (int k = start + 1; k <= end; k++)
   {
      if (a[k] < a[index])
         index = k;
   }
   return index;
}
```

(b) Return index of last occurrence:

```
int findMinimum(int[] a, int start, int end)
{
   int index = start;
   for (int k = start + 1; k <= end; k++)
   {
      if (a[k] <= a[index])
         index = k;
   }
   return index;
}
```

The only difference is the use of `<=` instead of `<` in the comparison of array elements.

(c) Return the minimum value itself. It is the simplest algorithm if the position is not required.

```
int findMinimum(int[] a, int start, int end)
{
   int min = a[start];
   for (int k = start + 1; k <= end; k++)
   {
      if (a[k] < min)
         min = a[k];
   }
   return min;
}
```

To obtain a `findMaximum` method for finding the maximum simply reverse the inequality in the if-statement.

■ EXAMPLE 12.1 **(Using `findMinimum` method, versions (a) and (b))** The statements

```
int[] scores = {56, 32, 27, 98, 27, 57, 68, 28, 45, 65};
int pos = findMinimum(score, 0, score.length - 1);
System.out.println("Position of minimum is " + pos);
System.out.println("Minimum value is " + scores[pos]);
```

find the minimum value in the `scores` array. For version (a) the values printed are `2` for `pos` and `27` for the minimum value. For version (b) the value 4 instead of `2` for `pos` is printed. This method can be tested in BeanShell by entering the method into the workspace editor and choosing ”Eval in workspace”. ■

We have used an array of integers here although an array of any type for which elements can be compared can be used. For example, to find the minimum balance in an array of `BankAccount` objects and return the index of its first occurrence use the method

```
int findMinimumBalance(BankAccount[] a, int start, int end)
{
   int index = start;
   for (int k = start + 1; k <= end; k++)
   {
      if (a[k].getBalance() < a[index].getBalance())
         index = k;
   }
   return index;
}
```

12.3 Running time of an algorithm

In order to compare two algorithms to see if one is more efficient than the other we need to measure the running time of each algorithm. One way to do this is to implement the algorithms and write a program that calculates their running time for various arrays and array sizes. The problem here is that the running time will depend on the particular hardware and software (processor, operating system, compiler, language). Faster computers or more efficient compilers will give lower running times.

We need a hardware/software-independent theoretical way to measure the running time of an algorithm in terms of the size of the problem and the number of times selected statements in the algorithm are executed.

For example, the running time of `findMinimum` depends on the number of array elements that are searched to find the minimum value. It is clear that finding the minimum in a million element array will take longer than for a 10 element array. If you time `findMinimum` for various array sizes, n, you will find that as n increases the running time increases linearly with n: searching a 10000 element array takes about twice as long as a 5000 element array which takes about twice as long as a 2500 element array, and so on. We say that `findMinimum` is a linear algorithm.

This means that for large n the running time will have the form $T(n) = an + b$ for some constants a and b. If n is large we can omit b in comparison to an. It is the fact that $T(n)$ is proportional to n, for large n, that is important here, not the constants a and b which depend on the particular hardware/software environment. To remove this dependency on a and b we simply say that all algorithms whose running time is at most $an + b$, such as `findMinimum`, are of order n and we write $T(n) = \mathcal{O}(n)$ to indicate that an upper bound on the running time is proportional to n. This is often called the "Big Oh" notation.

To derive this result mathematically we need some representative statements to count in the `findMinimum` algorithm. In this way we reduce the running time calculation to a counting problem. For example, we can use the number of times the if-statement inside the for loop is executed (the number of comparisons). This is the number of times the for loop is executed, namely $end - start + 1$, and this is just the number, n, of elements in the subarray. Therefore $T(n) = n$. We could choose other measures such as the total number of comparisons and the total number of assignment statements but then we would have difficulty calculating the exact number of operations since the number of times the assignment statement inside the if-statement is executed depends on the elements in the array. In any case this number would still have an upper bound of the form $an + b$ so $T(n) = \mathcal{O}(n)$ although this is more difficult to show.

We can also define the **best**, **average**, and **worst** case behavior of an algorithm. For the linear search algorithm each can be shown to be $\mathcal{O}(n)$. For other algorithms the worst case behavior is often easy to determine but the average behavior can be difficult since it depends on the probabilistic distribution of the elements in the array. In any case the worst case behavior gives an upper bound on the running time.

■ $\boxed{\text{EXAMPLE 12.2}}$ **(Mathematical definition of $\mathcal{O}$)** We can give a precise definition of the $\mathcal{O}$ notation as follows: Given two functions f and g we say that $f(n) = \mathcal{O}(g(n))$ if there exist constants $c > 0$ and $N > 0$ such that $0 \le f(n) \le cg(n)$ for all $n > N$. It is important to realize that $\mathcal{O}(n)$ is not a function. It represents an infinite set of functions. Therefore you should really interpret

$T(n) = O(n)$ to mean $T(n) \in O(n)$. There are other measures of the rate of growth of a function denoted by $\Omega(g(n))$, for a lower bound, and $\Theta(g(n))$, for both upper and lower bounds, that we will not consider here. ■

■ $\boxed{\text{EXAMPLE 12.3}}$ **(Example of O calculations)** It is easy to show that $an+b$ is $O(n)$. Let $c = a+1$. Then $an+b \le cn$ if $b \le n$. Therefore let N be any integer greater than b and we will have $an+b \le cn$ whenever $n > N$. This shows for example that n, $2n$ $3n+1/n$ are all $O(n)$. ■

■ $\boxed{\text{EXAMPLE 12.4}}$ **(Example of O calculations)** Let us show that n^2+3n+1 is $O(n^2)$. Let $c = 2$. Then $n^2+3n+1 \le 2n^2$ if $n^2 \ge 3n+1$ which is true if $n > 4$. Therefore let $N = 4$ and n^2+3n+1 is $O(n^2)$ for all $n > 4$. ■

12.4 Searching algorithms

The linear search algorithms was briefly discussed in Chapter 8, Section 8.4. Here we consider the linear search algorithm and both recursive and non-recursive versions of the much superior binary search algorithm and we obtain upper bounds on their running times. We will develop these algorithms for arrays of type `int[]` although it is easy to modify them to search other types of arrays.

The Java method for each of these three algorithms has a prototype of the form

```
public static int search(int[] a, int x, int start, int end)
```

where `a` is the array to search, `x` is the element to search for in the subarray defined by `start` and `end`, and the return value is either the position at which `x` is found or -1 if `x` is not found.

12.4.1 Linear search algorithm

The `findMinimum` and `findMaximum` algorithms are examples of search algorithms since we are searching for the smallest or largest value. An important variation is to search for a given value. We now generalize the results in Chapter 8, Section 8.4, so that the linear search is applied to a subarray rather than the entire array.

In a linear search of a subarray we are looking for a given value x among the array elements in the subarray. If we find it then we can return the array index at which it is found, otherwise we can return the invalid index value -1. The linear search problem can be stated as follows:

> "Given the array $\langle a_0, \ldots, a_{n-1} \rangle$, the index values *start* and *end* defining a subarray $\langle a_{start}, \ldots, a_{end} \rangle$, and a value x to find, determine an index i such that $a_i = x$ and $start \le i \le end$. If such an index cannot be found let the index be -1."

A while loop is appropriate here since we do not know how many times the body of the loop will be executed. We need to stop executing the body if the element we are looking for is found. The loop continues as long as we are within the subarray and as long as we have not found the element we are looking for. The pseudo-code algorithm is given in Figure 12.2. There are two ways the while loop can terminate. If $index \le end$ is false then we have "gone off the end" of the subarray

```
**ALGORITHM** LinearSearch($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, $x$, $start$, $end$)
$index \leftarrow start$
**WHILE** $index \leq end \wedge a_{index} \neq x$ **DO**
    $index \leftarrow index + 1$
**END WHILE**
**IF** $index > end$ **THEN**
    **RETURN** $-1$
**ELSE**
    **RETURN** $index$
**END IF**
```

Figure 12.2: Pseudo-code linear search algorithm

```
**ALGORITHM** LinearSearch($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, $x$, $start$, $end$)
$index \leftarrow start$
**WHILE** $index \leq end$ **DO**
    **IF** $a_{index} = x$ **THEN**
        **RETURN** $index$
    **END IF**
    $index \leftarrow index + 1$
**END WHILE**
**RETURN** $-1$
```

Figure 12.3: Alternate Pseudo-code linear search algorithm

and the entire boolean expression is false so the loop will exit. The expression $a_{index} \neq x$ will not be evaluated in this case, assuming short-circuit evaluation as in Java (in an expression like $a \wedge b$ if a is false b is never evaluated). Otherwise the array index could be out of range. If the element x is found then the expression $a_{index} \neq x$ will be false and the loop will exit. When the loop exits we can test *index* to see which exit was taken. If $index > end$ then we did not find x so -1 is returned. Otherwise x was found and *index* is returned.

An alternate version that may be easier to understand is shown in Figure 12.3. As for the `findMinimum` method there are other variations. We could run the loop backwards and find the last occurrence of `x` rather than the first occurrence. Or, if the position is not required, a `boolean` return type can have the value true if `x` is found and false otherwise.

Order of linear search

Since the while loop exits immediately if x is found, the best case behavior will occur if x is the first element in the subarray. The running time will not depend at all on the size, n, of the array. To indicate that the running time does not depend on n we write $T(n) = O(1)$.

Step	**Left subarray**	**Middle value**	**Right subarray**
1	3 5 7 11	21	47 56 63 84 89
2	47 56	63	84 89
3	empty	47	56
4	empty	56	empty

Table 12.1: Binary search example

The worst case behavior will occur if x is found at the last position in the subarray or x is not found. In this case every element in the subarray is examined. Defining $n = end - start + 1$, the body of the while loop will be executed n times. Therefore the worst case behavior is $T(n) = O(n)$. The average case behavior will depend on the values of the array elements and a probabilistic argument shows that $T(n) = O(n)$ in this case too.

12.4.2 Recursive binary search algorithm

The linear search algorithm is an $O(n)$ algorithm so it is not useful for large n. If we can assume that the array elements are sorted in increasing order (or decreasing order) then we can write a much better algorithm called the binary search algorithm because it continually divides the problem size (number of elements to search) in half until the element is found. This division process is often called bisection. On the other hand each iteration of the linear search algorithm would eliminate only one element.

To develop the recursive algorithm we start with an array $\langle a_0, \ldots, a_{n-1} \rangle$ sorted in increasing order. The half to be searched at each step will be a subarray of the form

$$\langle a_{start}, \ldots, a_{end} \rangle, \text{ where } a_{start} \le a_{start+1} \le \cdots \le a_{end}$$

We want to determine if x is found in the subarray. To begin the bisection process we look at the middle element, a_{mid}, of the subarray which we define by the index $mid = (start + end)/2$. This will be the middle element if there is an odd number of elements in the subarray and the leftmost of the two middle elements if there is an even number of elements. There are two base cases and two recursive cases to consider:

1. If $start > end$, the subarray is empty and x is not found (base case).
2. If $x = a_{mid}$, then x has been found (base case).
3. If $x < a_{mid}$, search left subarray $\langle a_{start}, \ldots, a_{mid-1} \rangle$ for x (recursive case).
4. If $x > a_{mid}$, search right subarray $\langle a_{mid+1}, \ldots, a_{end} \rangle$ for x (recursive case).

As an example, lets try to find 56 in the ten element array $\langle 3, 5, 7, 11, 21, 47, 56, 63, 84, 89 \rangle$. The bisection results are shown in Table 12.1. The important feature of this algorithm is that the number of elements remaining to be searched is cut in half at each step.

The running time is proportional to the total number of bisections that are made and this is proportional to the number of comparisons of x with a_{mid}. The best case behavior occurs when we

n	$\log_2 n = (\ln n/\ln 2)$
10	3
100	7
1000	10
10^6	20
10^9	30
10^{40}	133

Table 12.2: Comparison of n and $\log_2 n$

are searching for the element at the middle. Then the algorithm is $O(1)$. In the worst case, which we now consider, the element is either found at the last bisection step, when both left and right subarrays are empty, or it is not found.

For a 10 element array there are never more than 4 bisections: the number of elements remaining to search at each step is $10 \to 5 \to 3 \to 2 \to 1$. For linear search there could be as many as 10 comparisons. For a 1000 element array the number of elements remaining to be searched at each step is

$$1000 \to 500 \to 250 \to 125 \to 63 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$

so there are never more than 10 bisections versus as many as 1000 comparisons for linear search. Similarly, for a 1,000,000 element array at most 20 bisections are needed versus as many as 1,000,000 comparisons for linear search. It is clear that binary search is far superior to linear search. Later we will show that the order of binary search is $O(\log_2 n)$. Table 12.2 shows how much faster n increases compared to $\log_2 n$: if it took 30 units of time to search a billion element array then a linear search could take as many as 10^9 units of time. Calculators usually have buttons for the base 10 logarithms, $\log_{10} n$, or base e logarithms, $\log_e n = \ln n$, but it is easy to calculate logarithms in other bases using the identity

$$\log_b x = \log_c x/\log_c b$$

Therefore, using $b = 2$, $c = e$, and $x = n$ we have $\log_2 n = \log_e n/\log_e 2 = \ln n/\ln 2$ as shown in the table.

We can now write a pseudo-code binary search algorithm. The bisection process terminates when the indices *start* and *end* satisfy $start > end$. This corresponds to an empty subarray. The algorithm is given in Figure 12.4. Recalling that recursion is a problem solving technique for which a problem is solved in terms of one or more smaller versions of itself and one or more non-recursive base cases, we see that this algorithm is naturally recursive: the problem of searching a subarray is expressed in terms of the two smaller problems of searching either the left half subarray or the right half subarray. The base case occurs when the subarray is empty and this occurs when $start > end$.

12.4.3 Non-recursive binary search algorithm

It is also easy to develop a non-recursive version of binary search. For a subarray $\langle a_{start}, \ldots, a_{end} \rangle$ we need to keep two indices *low* and *high* that define the half $\langle a_{low}, \ldots, a_{high} \rangle$ to be searched next.

```
**ALGORITHM** BinarySearch($\langle a_{start}, \ldots, a_{end} \rangle$, $x$)
**IF** $start \leq end$ **THEN**
    $mid \leftarrow (start + end)/2$
    **IF** $x = a_{mid}$ **THEN**
        **RETURN** $mid$
    **ELSE IF** $x < a_{mid}$ **THEN**
        **RETURN** BinarySearch( $\langle a_{start}, \ldots, a_{mid-1} \rangle$, $x$)
    **ELSE**
        **RETURN** BinarySearch( $\langle a_{mid+1}, \ldots, a_{end} \rangle$, $x$)
    **END IF**
**END IF**
**RETURN** $-1$ // $x$ not found
```

Figure 12.4: Pseudo-code recursive binary search algorithm

1	3	5	7	11	21	47	56	63	84	89
$low = 0$										$high = 10$
						$low = 6$				$high = 10$
						$low = 6$	$high = 7$			
							$low, high = 7$			

Table 12.3: Low and high indices for non-recursive binary search, showing how *low* moves to the right and *high* moves to the left.

Initially we let $low = start$ and $high = end$. Then mid is calculated and if the left half is chosen $high$ needs to be adjusted to $high = mid - 1$. If the right half is chosen them low needs to adjusted to $low = mid + 1$. As the bisection proceeds low moves to the right and $high$ moves to the left so we can use a while loop that continues as long as $low \leq high$. This is illustrated in Table 12.3 for finding 56 in the array $\langle 1, 3, 5, 7, 11, 21, 47, 56, 63, 84, 89 \rangle$ with $start = 0$ and $end = 10$.

A pseudo-code algorithm for the non-recursive binary search algorithm is shown in Figure 12.5.

12.4.4 Running time of binary search algorithm

The results in Table 12.2 suggest that the worst case behavior of binary search is $O(\log_2 n)$ (which is also $O(\log_b n)$ for any base b). We now sketch the proof using the recursive version. Let n be the number of elements to search and let $T(n)$ be the running time for n elements defined as the number of bisections required. This will be the number of times the if-statement is executed. To simplify the proof we also assume that n is a power of 2: $n = 2^m$ for some m. Then we can search a subarray of n elements using one bisection to determine which half to use and $T(n/2)$ bisections

```
ALGORITHM NRBinarySearch($\langle a_{start}, \ldots, a_{end} \rangle$, $x$)
$low \leftarrow start$
$high \leftarrow end$
WHILE $low \leq high$ DO
    $mid \leftarrow (start + end)/2$
    IF $x < a_{mid}$ THEN
        $high \leftarrow mid - 1$
    ELSE IF $x > a_{mid}$ THEN
        $low \leftarrow mid + 1$
    ELSE
        RETURN $mid$
    END IF
END WHILE
RETURN $-1$
```

Figure 12.5: Pseudo-code non-recursive binary search algorithm

to search this half. This gives the following recurrence relation connecting $T(n)$ and $T(n/2)$:

$$T(n) \quad = \quad T(n/2) + 1, \text{ with initial condition } T(1) = 1.$$

Substituting $n/2$ for n gives $T(n/2) = T(n/2^2) + 1$. Continuing

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/2^2) + 1 + 1 = T(n/2^2) + 2 \\ &= T(n/2^3) + 1 + 2 = T(n/2^3) + 3 \\ &\cdots \\ &= T(n/2^m) + m \\ &= T(1) + m \\ &= 1 + m \\ &= 1 + \log_2 n, \text{ since } n = 2^m \\ &= O(\log_2 n) \end{aligned}$$

Therefore binary search is an $O(\log n)$ algorithm and, as Table 12.2 shows, is much superior to linear search. If n is not a power of two we can use $\lceil n/2^k \rceil$ at each stage instead of $n/2^k$, noting that $2^k \leq n \leq 2^{k+1}$ for some k ($\lceil x \rceil$ denotes the smallest integer greater than or equal to x).

12.4.5 Class of static searching methods

The three search algorithms can easily be translated into Java. We place them in a class called `IntArraySearch` as static methods:

Class `IntArraySearch`

`book-project/chapter12/searching`

```
package chapter12.searching;

/**
 * Static methods for searching an integer array
 */
public class IntArraySearch
{
   /**
    * Search subarray for a given element using linear search algorithm.
    * @param a The array to search
    * @param x The value to search for
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    * @return If <code>x</code> is found the first array index such that
    *    <code>x = a[i]</code> else <code>-1</code> to indicate failure.
    */
   public static int linearSearch(int[] a, int x, int start, int end)
   {
      int i = start;
      while ((i <= end) && (a[i] != x))
         i++;
      if (i <= end)
         return i;
      else
         return -1;
   }

   /**
    * Search subarray for a given element using the recursive
    * binary search algorithm. It is assumed that the array is sorted
    * in increasing order.
    * @param a The array to search
    * @param x The value to search for
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    * @return If <code>x</code> is found the first array index such that
    *    <code>x = a[i]</code> else <code>-1</code> to indicate failure.
    */
   public static int rBinarySearch(int[] a, int x, int start, int end)
   {
      if (start <= end)
      {
         int mid = (start + end) / 2;
         if (x < a[mid])
            return rBinarySearch(a, x, start, mid - 1); // search a[start] to a[mid-1]
         else if (x > a[mid])
            return rBinarySearch(a, x, mid + 1, end); // search a[mid+1] to a[end]
         else
            return mid; // x found and x = a[mid]
      }
      return -1; // x not found
   }

   /**
    * Search a subarray for a given element using the non-recursive
    * binary search algorithm. It is assumed that the array is sorted
    * in increasing order.
    * @param a The array to search
    * @param x The value to search for
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    * @return If <code>x</code> is found the first array index such that
    *    <code>x = a[i]</code> else <code>-1</code> to indicate failure.
    */
   public static int nrBinarySearch(int[] a, int x, int start, int end)
   {
      int low = start;
      int high = end;
      while (low <= high)
      {
         int mid = (low + high) / 2;
         if (x < a[mid])
            high = mid - 1;  // search left half a[low] to a[high-1]
         else if (x > a[mid])
            low = mid + 1;   // search right half a[mid+1] to a[high]
         else
            return mid;      // x found and x = a[mid]
      }
      return -1; // x not found
   }
}
```

12.4.6 Testing the search algorithms

To test these classes we use the following `IntArraySearchTester` class:

Class `IntArraySearchTester`

book-project/chapter12/searching

```
package chapter12.searching;
import java.util.Scanner;

/**
 * Test the linear and binary searching algorithms.
 */
public class IntArraySearchTester
{
   /** Test the search algorithms.
    */
   public void doTest()
   {
      Scanner input = new Scanner(System.in);

      // read the array

      System.out.print("Enter number of integers in array: ");
      int size = input.nextInt(); input.nextLine();
      int[] testArray = new int[size];
      for (int k = 0; k < testArray.length; k++)
      {
         System.out.print("Enter element " + k + ": ");
         testArray[k] = input.nextInt(); input.nextLine();
      }

      // Read element to find and the subarray start, end indices

      System.out.print("Enter element to find: ");
      int x = input.nextInt(); input.nextLine();
      System.out.print("Enter start index for subarray: ");
      int start = input.nextInt(); input.nextLine();
      System.out.print("Enter end index for subarray: ");
      int end = input.nextInt(); input.nextLine();

      // Search the array and display results for each algorithm

      int pos;
      pos = IntArraySearch.linearSearch(testArray, x, start, end);
      displayResult(pos, x);
      pos = IntArraySearch.rBinarySearch(testArray, x, start, end);
      displayResult(pos, x);
      pos = IntArraySearch.nrBinarySearch(testArray, x, start, end);
      displayResult(pos, x);
   }

   public void displayResult(int pos, int x)
   {
      if (pos < 0)
         System.out.println("Element " + x + " was not found");
      else
         System.out.println("Element " + x + " was found at position " + pos);
   }

   public static void main(String[] args)
   {
      new IntArraySearchTester().doTest();
   }
}
```

12.5 Sorting algorithms

We now consider four well-known sorting algorithms applied to a subarray of an integer array. The first two, selection sort and insertion sort, are quadratic algorithms. Their worst and average case behavior is $O(n^2)$. The other two, quicksort and mergesort, are much superior for large arrays since they have average case behavior $O(n\log n)$. Mergesort also has worst case behavior $O(n\log n)$ but requires a temporary array for storage, and quicksort has worst case behavior $O(n^2)$ but requires no temporary array.

We design our algorithms to sort arrays of integers in increasing order. For example, the array $\langle 5,3,8,5,4,2,2\rangle$ is not sorted. In increasing order the sorted array is $\langle 2,2,3,4,5,5,8\rangle$. Later we show how to do generic sorting using arrays of type `Object[]`.

The Java methods for each of these four algorithms has a prototype of the form

```
public static void sort(int[] a, int start, int end)
```

where `a` is the array $\langle a_0,\ldots,a_{n-1}\rangle$ and the subarray $\langle a_{start},\ldots,a_{end}\rangle$ to sort is defined by `start` and `end`.

12.5.1 Selection sort algorithm

The selection sort algorithm is one of the easiest sorting algorithms to understand because of its intuitive nature. We apply it to the subarray $\langle a_{start},\ldots,a_{end}\rangle$ of the array $\langle a_0,\ldots,a_{n-1}\rangle$ whose elements have an order defined for them. The algorithm for increasing order is

1. Find the smallest of the elements in $\langle a_{start},\ldots,a_{end}\rangle$ and exchange (swap) it with the element a_{start} at position $start$. Now a_{start} is the smallest element in the subarray and it is in the correct position.

2. Find the smallest of the elements in the remaining subarray $\langle a_{start+1},\ldots,a_{end}\rangle$ and exchange (swap) it with the element $a_{start+1}$ at position $start+1$. Now a_{start} and $a_{start+1}$ are the two smallest elements in the original subarray and they are in the correct position ($a_{start} \le a_{start+1}$).

3. Repeat, using smaller subarrays until the last subarray to sort is $\langle a_{end-1},a_{end}\rangle$.

This gives the top level pseudo-code algorithm

```
FOR $i \leftarrow start$ TO $end-1$ DO
    Find the index $k$ of the smallest element in subarray $\langle a_i,\ldots,a_{end}\rangle$
    Exchange (swap) elements at positions $i$ and $k$.
END FOR
```

As an example, consider the array $\langle a_0,\ldots,a_7\rangle$ given by $\langle 44,55,12,42,94,18,6,67\rangle$ and use $start = 0$, $end = 7$. The steps are shown in Table 12.5 where underlined elements are in their correct position. The complete pseudo-code algorithm for selection sort is given in Figure 12.6.

Note carefully in the outer for loop that the last value of i is $end-1$ since the last subarray to examine is $\langle a_{end-1},a_{end}\rangle$, but in the inner for loop the last value of j is end since we must search

Step	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	**operation**
1	44	55	12	42	94	18	6	67	swap 6 with 44
2	**6**	55	12	42	94	18	44	67	swap 12 with 55
3	**6**	**12**	55	42	94	18	44	67	swap 18 with 55
4	**6**	**12**	**18**	42	94	55	44	67	swap 42 with itself
5	**6**	**12**	**18**	**42**	94	55	44	67	swap 44 with 94
6	**6**	**12**	**18**	**42**	**44**	55	94	67	swap 55 with itself
7	**6**	**12**	**18**	**42**	**44**	**55**	94	67	swap 67 with 94)
8	**6**	**12**	**18**	**42**	**44**	**55**	**67**	**94**	(done)

Table 12.4: Selection sort example

```
ALGORITHM selectionSort($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, $start$, $end$)
FOR $i \leftarrow start$ TO $end - 1$ DO
    $k \leftarrow i$
    FOR $j \leftarrow i + 1$ TO $end$ DO
        IF $a_j < a_k$ THEN
            $k \leftarrow j$
        END IF
    END FOR
    $temp \leftarrow a_k$
    $a_k \leftarrow a_i$
    $a_i \leftarrow temp$
END FOR
```

Figure 12.6: Pseudo-code selection sort algorithm

each subarray until the last element. Also, three assignment statements are needed to exchange (swap) two values, since it is necessary to use a temporary variable to save the first element of the pair being swapped.

As we did for the searching algorithms, it is easy to translate this pseudo-code algorithm into the following static method called `selectionSort`.

```
public static void selectionSort(int[] a, int start, int end)
{
   for (int i = start; i < end; i++)
   {
      // find position k of minimum element among
      // the elements a[i] to a[end]

      int k = i;
      for (int j = i+1; j <= end; j++)
      {
         if (a[j] < a[k])
            k = j;
      }

      // swap the smallest element found (it’s a[k]) with a[i]

      int temp = a[k];
      a[k] = a[i];
      a[i] = temp;
   }
}
```

This sort method is placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

12.5.2 Running time for selection sort

One measure of the running time is the number of times the if statement is executed in the inner loop. This is the number of times the loop is executed and this in turn depends on the outer loop index. We can assume that there are n array elements so $n = end - start + 1$. We can use the results in Table 12.5 to do the counting. Adding the entries in the last column gives the running time

$$T(n) \;=\; n-1+n-2+n-3+\cdots+1 \;=\; 1+2+3+\cdots+n-1.$$

If you don’t know the formula for this common sum write it twice, once forward and once backward, as follows

$$\begin{array}{rcccccccc} T(n) & = & 1 & + & 2 & + & 3 & +\cdots+ & n-1 \\ T(n) & = & n-1 & + & n-2 & + & n-3 & +\cdots+ & 1. \end{array}$$

Now add to obtain $2T(n) = n+n+\ldots+n = n(n-1)$ since n is repeated $n-1$ times. Therefore

$$T(n) \;=\; \frac{n(n-1)}{2} \;=\; \frac{1}{2}n^2 - \frac{1}{2}n \;=\; O(n^2).$$

outer loop index	**inner loop index**	**inner loop executions**
$start$	$start+1,\ldots,end$	$end-start=n-1$
$start+1$	$start+2,\ldots,end$	$end-start-1=n-2$
$start+2$	$start+3,\ldots,end$	$end-start-2=n-3$
$\ldots$	$\ldots$	$\ldots$
$end-1$	$end,\ldots,end$	1

Table 12.5: Counting inner loop executions for selection sort

Since the inner and outer for loop indices do not depend on the actual array elements, according to our definition of the running time, the best, average, and worst case behavior of selection sort are all $O(n^2)$. Of course the actual running time is sensitive to the distribution of elements in the array since this affects the number of times the assignment statement is executed in the if statement.

12.5.3 Insertion sort algorithm

Insertion sort is another intuitive algorithm and is performed by poker or bridge players when they arrange their cards in order. To understand this algorithm think of dividing the subarray to be sorted into two subarrays, a left one called L which is already sorted and a right one called R which is unsorted. If the initial subarray is $\langle a_{start},\ldots,a_{end}\rangle$ then its two parts are

$$L = \langle a_{start}\rangle, \quad R = \langle a_{start+1},\ldots,a_{end}\rangle$$

since a one-element part is always sorted. We begin by taking the first element of R and moving it to its proper place in the left part to give

$$L = \langle a_{start}, a_{start+1}\rangle, \quad R = \langle a_{start+2},\ldots,a_{end}\rangle$$

where $a_{start} \le a_{start+1}$. Next we move $a_{start+2}$ to its proper place in L. After several steps we arrive at the general situation

$$L = \langle a_{start},\ldots a_{i-1}\rangle, \quad R = \langle a_i,\ldots,a_{end}\rangle$$

for which $a_{start} \le a_{start+1} \le \cdots \le a_{i-1}$. Rather than moving a_i left, by swapping, until it is in its proper position in L we can move a_{i-1} to the right one place if it is larger than a_i, a_{i-2} to the right one place if it is larger than a_i and so on until we obtain a “hole” in L into which we can drop a_i. This “move and drop” technique is more efficient than swapping since it involves less data movement.

The following example illustrates this for the array $A=\langle 44,55,12,42,94,18,6,67\rangle$. Starting with $L=\langle 44\rangle$ and $R=\langle 55,12,42,94,18,6,67\rangle$ we see that 55 is already in its correct position with respect to 44 so the next step is $L=\langle 44,55\rangle$ and $R=\langle 12,42,94,18,6,67\rangle$. Continuing we obtain Figure 12.7. The vertical bars indicate the division between the subarrays L and R at each step. The complete pseudo-code algorithm for insertion sort is given in Figure 12.8. At step i the sorted part is $L=\langle a_{start},\ldots,a_{i-1}\rangle$ and the unsorted part is $R=\langle a_i,\ldots,a_{end}\rangle$. In this step a hole for $x=a_i$ is opened up by moving a_{i-1}, $a_{i-2},\ldots$ to the right in the while loop if they are greater than x.

We can translate this pseudo-code algorithm to the following method called `insertionSort` which will later be placed in the `IntArraySort` class along with the other sorting algorithm.

step				**array**					**operation**
	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	
1	44 |	55	12	42	94	18	6	67	put 55 in L
2	44	55 |	12	42	94	18	6	67	put 12 in L
3	12	44	55 |	42	94	18	6	67	put 42 in L
4	12	42	44	55 |	94	18	6	67	put 94 in L
5	12	42	44	55	94 |	18	6	67	put 18 in L
6	12	18	42	44	55	94 |	6	67	put 6 in L
7	6	12	18	42	44	55	94 |	67	put 67 in L
8	6	12	18	42	44	55	67	94	done

Figure 12.7: An insertion sort example for $\langle 44, 55, 12, 42, 94, 18, 6, 67 \rangle$.

```
ALGORITHM insertionSort($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, $start$, $end$)
FOR $i \leftarrow start + 1$ TO $end$ DO
    $x \leftarrow a_i$
    $j \leftarrow i - 1$
    WHILE $j \geq start \wedge x < a_j$ DO
        $a_{j+1} \leftarrow a_j$
        $j = j - 1$
    END WHILE
    $a_{j+1} \leftarrow x$
END FOR
```

Figure 12.8: Pseudo-code insertion sort algorithm

```
public static void insertionSort(int[] a, int start, int end)
{
   for (int i = start+1; i <= end; i++)
   {
      // Sorted part of array is a[start], ..., a[i-1]
      // Unsorted part is a[i], ..., a[end]

      int x = a[i]; // left element of unsorted part
      int j = i-1;  // right index of sorted part

      // move elements right until position for x is found.

      while ( (j >= start) && (x < a[j]) )
      {
         a[j+1] = a[j]; // move a[j] one place to the right
         j--;
      }
```

outer loop index	**inner loop index**	**inner loop executions**
$start+1$	$start,\ldots,start$	1
$start+2$	$start,\ldots,start+1$	2
$start+3$	$start,\ldots,start+1$	3
$\ldots$	$\ldots$	$\ldots$
end	$start,\ldots,end-1$	$end-start$

Table 12.6: Counting inner loop executions for worst case insertion sort

```
            a[j+1] = x; // drop x into the hole found
        }
    }
```

This sort method is placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

12.5.4 Running time for insertion sort

For this algorithm let us use the number of times the statements inside the while loop are executed to determine the running time. This is a more difficult problem than for selection sort since the number of times the while loop is executed depends on the array. Therefore we will calculate the worst case behavior. The inner loop will be executed the maximum number of times if j is decremented to the beginning of the array each time. This occurs when the array is sorted in reverse order. For example, $\langle 10,9,8,7,6,5,4,3,2,1\rangle$. In this case we obtain the results in Table 12.6, again using $n = end - start + 1$. Adding the entries in the last column gives

$$T(n) \;=\; 1+2+\cdots+end-start \;=\; \frac{n(n-1)}{2} \;=\; O(n^2).$$

Clearly the worst case running time is $O(n^2)$. It can be shown that the average time is also $O(n^2)$. The best case behavior occurs when the initial array is already sorted. Then the statements in the while loop are never executed. Since the outer loop is executed $n-1$ times the best case behavior is $O(n)$.

12.5.5 Simulation to compare selection and insertion sort

Simulations are important in the analysis of algorithms and computer systems since it is not always possible to obtain good theoretical estimates. Also it may be necessary to optimize an algorithm for a particular computer system. Even though we can analyze selection sort and insertion sort and determine their average behavior is $O(n^2)$, it is useful to compare them on a given computer system to see which is faster. Since the worst case behavior of insertion sort compares with the average case behavior of selection sort we expect that insertion sort is a little faster even though both algorithms are $O(n^2)$ (a more careful theoretical analysis confirms this).

It is easy to write a simulation program to estimate the running time of a sorting algorithm. We simply generate random arrays and average the times over a certain number of trials. We can use the `Random` class in `java.util` to generate random integers. For example, if we define

```
Random rand = new Random(1234);
```

then each call `rand.nextInt()` generates the next random number in the sequence determined by the seed `1234`. Each seed generally produces a different sequence of integers. To time the selection sort algorithm we can use the statements

```
long startTime = System.nanoTime();
IntArraySort.selectionSort(array, 0, array.length - 1);
long endTime = System.nanoTime();
double sortTime = (endTime - startTime) / 1E9; // seconds
```

Here is a class that can be used to time the selection sort and insertion sort algorithms algorithm. Note that we have run one extra trial and not counted the first one since the Java compiler will do some ”just in time” compiling after the first run of a block of code:

Class `QuadraticSortTimer`

`book-project/chapter12/sorting`

```
package chapter12.sorting;
import java.util.Scanner;
import java.util.Random;

/**
 * Estimate the average time of the selection and insertion
 * sort algorithms using a given array size and number of trials
 * to average
 */
public class QuadraticSortTimer
{
   /**
    * Perform a sorting simulation to compare insertion
    * sort and selection sort and display results.
    */
   public void doTest()
   {
      Scanner input = new Scanner(System.in);
      String sortType = "";

      // Get the sorting algorithm

      System.out.println("Selection sort: 1");
      System.out.println("Insertion sort: 2");
      System.out.println("Enter 1 or 2");
      int choice = input.nextInt(); input.nextLine();

      // Get size of array and number of trials to average

      System.out.print("Enter size of array: ");
      int size = input.nextInt(); input.nextLine();
      System.out.print("Enter number of trials to average: ");
```

```
int numTrials = input.nextInt(); input.nextLine();
Random rand = new Random(1234);
int[] array = new int[size];

double timeSum = 0.0;
for (int trial = 1; trial <= numTrials + 1; trial++)
{
   // fill array with random integers

   for (int k = 0; k < array.length; k++)
      array[k] = rand.nextInt();

   // compute the time for this trial

   long endTime = 0L, startTime = 0L;
   if (choice == 1)
   {
      sortType = "Selection sort";
      startTime = System.nanoTime(); // nanoseconds
      IntArraySort.selectionSort(array, 0, array.length - 1);
      endTime = System.nanoTime();
   }
   else
   {
      sortType = "Insertion sort";
      startTime = System.nanoTime(); // nanoseconds
      IntArraySort.insertionSort(array, 0, array.length - 1);
      endTime = System.nanoTime();
   }

   if (trial > 1) // skip first trial (do numTrials trials)
   {
      double sortTime = (endTime - startTime)/ 1E9; // seconds
      System.out.println("Sort time: " + sortTime + " seconds");
      timeSum = timeSum + (double) sortTime;
   }
}

// Compute average time over all trials

double averageTime = timeSum / (double) numTrials;

// Display the results

System.out.println();
System.out.println(sortType);
System.out.println("Array size (n): " + size);
System.out.println("Number of trials: " + numTrials);
System.out.println("Average running time: " + averageTime + " seconds");
double nSquared = (double) size * (double) size;
double ratio = averageTime / nSquared;
System.out.println("average / n^2: " + ratio);
```

size n	**selection sort**	$T(n)/n^2$	**insertion sort**	$T(n)/n^2$
5,000	0.41 sec	1.64×10^{-8}	0.25 sec	1.01×10^{-8}
10,000	2.07 sec	2.07×10^{-8}	1.02 sec	1.02×10^{-8}
20,000	8.50 sec	2.12×10^{-8}	4.36 sec	1.09×10^{-8}
30,000	19.5 sec	2.16×10^{-8}	10.0 sec	1.11×10^{-8}
40,000	34.7 sec	2.17×10^{-8}	18.8 sec	1.17×10^{-8}
50,000	53.6 sec	2.15×10^{-8}	29.0 sec	1.16×10^{-8}
100,000	213.0 sec	2.13×10^{-8}	125.0 sec	1.25×10^{-8}

Table 12.7: Comparing average times for selection sort and insertion sort.

```
   }

   public static void main(String[] args)
   {
      QuadraticSortTimer timer = new QuadraticSortTimer();
      timer.doTest();
   }
}
```

This class can be used to calculate the average running time of these algorithms. This is done by generating random arrays of integers, calculating the time it takes to sort them, adding up the times and dividing by the total number of trials ,and displaying the final result for the average time.

Since the $O(n^2)$ algorithms have running time proportional to n^2 for large n it would also be useful to estimate the constant a such that

$$T(n) = an^2, \text{ approximately for large } n.$$

We can do this by displaying the ratio $T(n)/n^2$, which should be a good estimate of the constant a for large n.

Some results of running this programs on a particular computer system are shown in Table 12.7. To estimate the average running time 10 trials were averaged. If we didn't know that both of these algorithms were $O(n^2)$ we could use this data to check it empirically. For selection sort the ratio $T(n)/n^2$ varies from 1.64×10^{-8} to 2.13×10^{-8}. These values are approximately constant so we have the empirical formula

$$T(n) = 2.1\times10^{-8}\, n^2, \text{ approximately for large } n$$

using 2.1×10^{-8} as an estimate for the constant a. Similarly for insertion sort the ratio $T(n)/n^2$ varies from 1.01×10^{-8} to 1.25×10^{-8}. This gives the empirical formula

$$T(n) = 1.2\times10^{-8}\, n^2, \text{ approximately for large } n$$

In this range insertion sort is almost twice as fast as selection sort on this computer system.

12.5.6 Mergesort

Mergesort is a naturally recursive algorithm for sorting a subarray by dividing it into two halves. If the initial subarray is $\langle a_{start}, \ldots, a_{end} \rangle$ then the two halves are

$$L = \langle a_{start}, \ldots, a_{mid} \rangle, \quad R = \langle a_{mid+1}, \ldots, a_{end} \rangle$$

where $mid = (start + end)/2$. Like the recursive binary search algorithm it is a "divide and conquer" algorithm. We apply mergesort to each half (smaller versions of the problem) and then we assume that there is some function called `merge` that will merge the two sorted halves together into a sorted subarray. The base case for the recursion occurs if $start = end$ since a one element subarray is already sorted. This gives the following simple Java method:

```
public static void mergeSort(int[] a, int start, int end)
{
   if (start == end)
      return; // one-element subarray is already sorted
   int mid = (start + end) / 2;
   mergeSort(a, start, mid); // mergesort the left half
   mergeSort(a, mid+1, end); // mergesort the right half
   merge(a, start, mid, end); // merge the two halves
}
```

Before we consider how to solve the merge problem let us assume that the `merge` method has been written and look at the recursive process. The two recursive calls to `mergeSort` simply divide the array into smaller and smaller subarrays until they have only one element. Then merge recombines each pair of sorted subarrays into a larger sorted subarray. An example for the array $\langle 8, 1, 6, 4, 10, 5, 3, 2, 22 \rangle$ is shown in Figure 12.9. The top half of the diagram shows how the recursive calls break the array into subarrays. The first value of *mid* is $(0+8)/2 = 4$ so we get the subarrays $\langle 8, 1, 6, 4, 10 \rangle$ and $\langle 5, 3, 2, 22 \rangle$. The second subarray is not generated until the left subarray has been fully subdivided in the left part of the diagram. The top half of the diagram shows how the recursive process divides the initial array into one element subarrays and the bottom half shows how the merge method recombines these subarrays to eventually produce the original array in sorted form. For example, the first merge step on the left combines the one-element subarrays $\langle 8 \rangle$ and $\langle 1 \rangle$ to obtain the sorted subarray $\langle 1, 8 \rangle$ and the last merge step combines the sorted subarrays $\langle 1, 4, 6, 8, 10 \rangle$ and $\langle 2, 3, 5, 22 \rangle$ to give the sorted array $\langle 1, 2, 3, 4, 5, 6, 8, 10, 22 \rangle$.

12.5.7 Merge algorithm for two sorted subarrays

The merge problem is interesting by itself. We can state it for arrays as follows:

> "Given a subarray $\langle a_{start}, \ldots, a_{end} \rangle$ partitioned by an index *split* with $start \le split \le end$, such that the subarrays $\langle a_{start}, \ldots, a_{split} \rangle$ and $\langle a_{split+1}, \ldots, a_{end} \rangle$ are each sorted in increasing order, sort the entire subarray $\langle a_{start}, \ldots, a_{end} \rangle$ into a temporary array $\langle t_0, \ldots, t_{end-start+1} \rangle$ such that $t_0 \le t_1 \le \ldots \le t_{end-start+1}$."

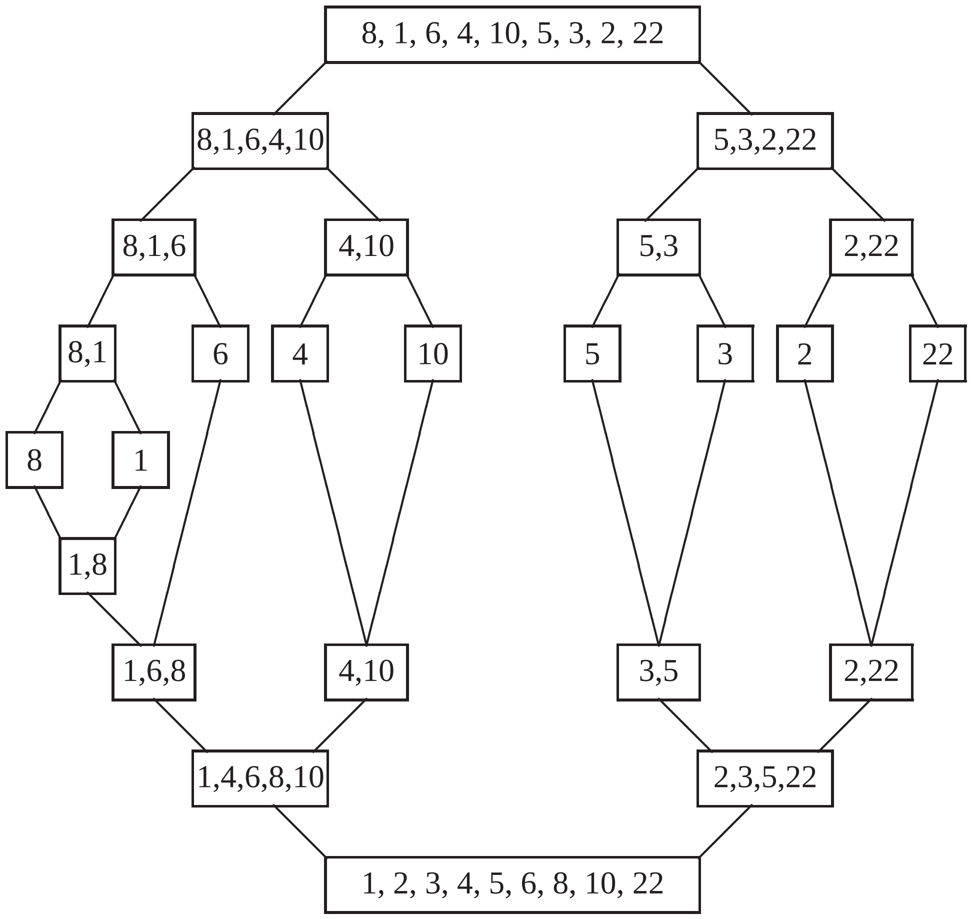


Figure 12.9: Mergesort example for the array $\langle 8, 1, 6, 4, 10, 5, 3, 2, 22 \rangle$

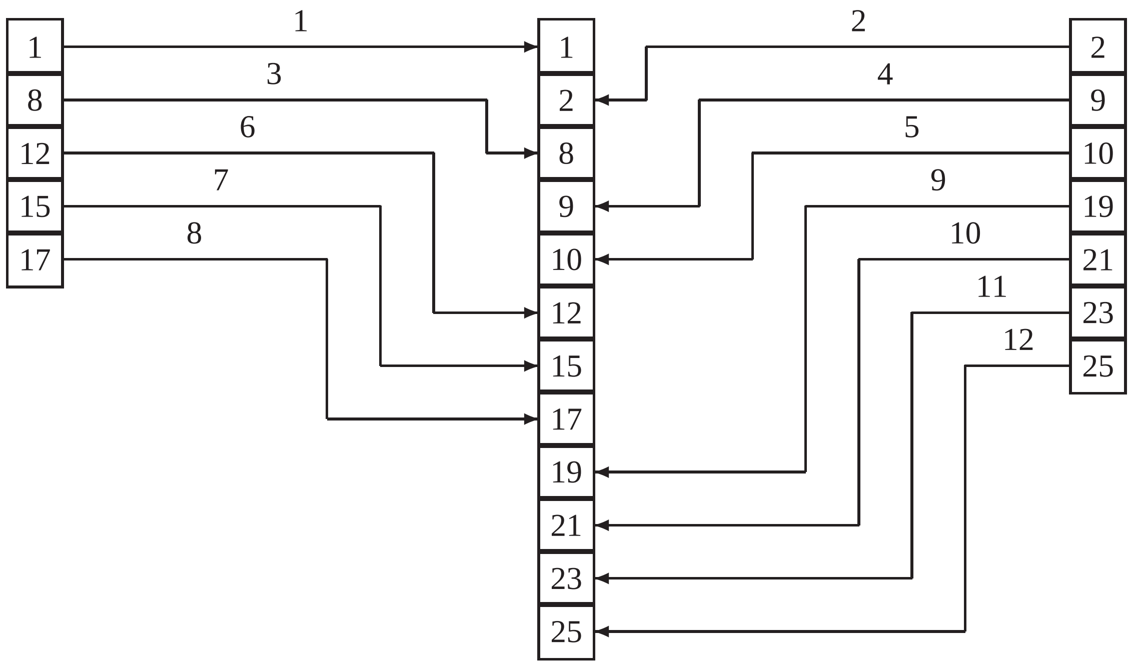


Figure 12.10: Merging the subarrays $\langle 1,8,12,15,17\rangle$ and $\langle 2,9,10,19,21,23,25\rangle$ to obtain the sorted subarray $\langle 1,2,8,9,10,12,15,17,19,21,23,25\rangle$.

For example, the unsorted array $\langle 1,8,12,15,17,2,9,10,19,21,23,25\rangle$ with $split = 4$ is such that the left subarray $\langle 1,8,12,15,17\rangle$ and the right subarray $\langle 2,9,10,19,21,23,25\rangle$ are each sorted. We want to obtain the sorted subarray $\langle 1,2,8,9,10,12,15,17,19,21,23,25\rangle$. The temporary array will be $\langle t_0,\ldots,t_{11}\rangle$. The merge process for this example is quite simple, as shown in Figure 12.10. Here the left subarray is shown vertically on the left and the right subarray is shown vertically on the right. The temporary array is shown in the center. The arrows indicate the data movement and the numbers above the horizontal lines indicate the step number.

The algorithm simply compares the current element from the left subarray with the current element from the right subarray and moves the smaller into the current position in the temporary array. First 1 is compared with 2 so 1 is moved into the first position in the temporary array. Then the next element, 8, of the first subarray is compared with 2 so 2 is moved into the next position in the subarray. This process continues until one or both subarrays are exhausted. In the example this occurs first for the left array at step 8. Then the remaining elements 19, 21, 23, and 25 in the right subarray are copied to the temporary array. Finally the temporary array is copied back to the original array. A top level pseudo-code algorithm is shown in Figure 12.11.

Since the first while loop will exhaust at least one of the left or right subarrays, only one of the remaining while loops will be executed to copy any remaining elements. The running time of merge is $O(n)$.

To write a Java method for this algorithm we need three index variables, `i` initialized to `start` indexes elements of the left subarray, `j` initialized to `split + 1` indexes elements of the right subarray, and `k` initialized to 0 indexes elements of the temporary array `t`. Assuming these initial-

```
ALGORITHM merge($\langle a_0, a_1, \ldots, a_{n-1} \rangle$, $start$, $split$, $end$)
$t \leftarrow \langle t_0, \ldots, t_{end-start+1} \rangle$
WHILE neither $\langle a_{start}, \ldots, a_{split} \rangle$ nor $\langle a_{split+1}, \ldots, a_{end} \rangle$ is exhausted DO
    Compare pairs of elements and copy smallest to temp array $t$
END WHILE
WHILE elements remain in $\langle a_{start}, \ldots, a_{split} \rangle$ DO
    Copy them to $t$
END WHILE
WHILE elements remain in $\langle a_{split+1}, \ldots, a_{end} \rangle$ DO
    Copy them to $t$
END WHILE
$\langle a_{start}, \ldots, a_{end} \rangle \leftarrow \langle t_0, \ldots, t_{end-start+1} \rangle$
```

Figure 12.11: Pseudo-code merge algorithm for two subarrays

izations the first while loop can be expressed as

```
while (i <= split && j <= end)
{
   if (a[i] < a[j]) // element in left subarray is smaller
   {
      t[k] = a[i];     // move it to temp array t
      i++;             // index next element in left subarray
   }
   else             // element in right subarray is smaller
   {
      t[k] = a[j];     // move it to temp array t
      j++;             // index next element in right subarray
   }
   k++;             // in either case index next position in t
}
```

After this loop executes either both subarrays are exhausted (if they have the same size) or there are elements in one of them that remain to be copied. The following loop will execute only if there are elements remaining in the left subarray:

```
while (i <= split)
{
   t[k] = a[i];
   k++;
   i++;
}
```

and the following loop will execute only if there are elements remaining in the right subarray:

```
while (j <= end)
```

```
{
   t[k] = a[j];
   k++;
   j++;
}
```

Finally the temporary array can be copied back to the original array using the for loop

```
for (k = 0; k < end - start + 1; k++)
   a[start + k] = t[k];
```

This can be done a little more efficiently using the special `arraycopy` method in the `System` class:

```
System.arraycopy(t, 0, a, start, end - start + 1);
```

Later the `mergeSort` and `merge` methods will be placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

```
public static void mergeSort(int[] a, int start, int end)
{
   if (start == end)
      return; // one-element subarray is already sorted
   int mid = (start + end) / 2;
   mergeSort(a, start, mid); // merge sort left half
   mergeSort(a, mid+1, end); // merge sort right half
   merge(a, start, mid, end); // merge the two halves
}

public static void merge(int[] a, int start, int split, int end)
{
   int n = end - start + 1; // number of elements to merge
   int[] t = new int[n];    // temporary storage required
   int i = start;           // index of elements in left subarray
   int j = split + 1;       // index of elements in right subarray
   int k = 0;               // index into temporary storage

   // merge elements from left and right subarray to temp array
   // until one or both of the subarrays are exhausted

   while (i <= split && j <= end)
   {
      if (a[i] < a[j]) // element in left subarray is smaller
      {
         t[k] = a[i];     // move it to temp array t
         i++;             // index next element in left subarray
      }
      else             // element in right subarray is smaller
      {
         t[k] = a[j];     // move it to temp array t
         j++;             // index next element in right subarray
      }
      k++;             // in either case index next position in t
   }

   // copy any remaining elements from left subarray to t

   while (i <= split)
   {
      t[k] = a[i];
      i++;
      k++;
   }

   // copy any remaining elements from right subarray to t

   while (j <= end)
   {
      t[k] = a[j];
      j++;
      k++;
   }

   // copy elements to a from temporary array t. Can also use
   // System.arraycopy(t, 0, a, start, end-start+1);

   for (k = 0; k < n; k++)
      a[start+k] = t[k];
}
```

If you like compact code the three loops in the `merge` method can be expressed as

```
while (i <= split && j <= end)
   if (a[i] < a[j]) t[k++] = a[i++];
   else t[k++] = a[j++]
while (i <= split) t[k++] = a[i++];
while (j <= end) t[k++] = a[j++];
for (k=0; k < n; k++) a[start+k] = t[k];
```

Here we make use of the fact that an expression like `a[i++]` means to first use `i` as an index and then increment it. Therefore an assignment such as

```
t[k++] = a[i++];
```

is equivalent to the three assignment statements

```
t[k] = a[i];
k = k + 1;
i = i + 1;
```

It should also be noted that `a[i++]` is quite different from `a[++i]` which increments `i` before using it as an array index.

12.5.8 Running time for mergesort

The calculation of the running time for mergesort is similar to the previous calculation for the recursive binary search. Let $T(n)$ be the running time for a subarray of size n. Then the `mergeSort` method shows that

$$T(n) = T(n/2) + T(n/2) + \text{ time to do merge}$$

The merge algorithm is $O(n)$ since each while loop executes in $O(n)$ time. The number of times each loop is executed depends only on the array size and not the actual array elements. Therefore let us assume that its running time is $an+b$ and that $n = 2^m$ for some m. Then

$$\begin{aligned}
T(n) &= 2T(n/2) + an + b \\
&= 2\left[2T(n/2^2) + a\frac{n}{2} + b\right] + an + b \\
&= 2^2T(n/2^2) + 2an + (1+2)b \\
&= 2^2\left[2T(n/2^3) + a\frac{n}{2^2} + b\right] + 2an + (1+2)b \\
&= 2^3T(n/2^3) + 3an + (1+2+2^2)b \\
&\cdots \\
&= 2^mT(n/2^m) + anm + (1+2+\cdots+2^{m-1})b \\
&= 2^mT(n/2^m) + anm + (2^m-1)b
\end{aligned}$$

Substituting $n = 2^m$ and using $T(1) = 1$ we obtain

$$\begin{aligned}
T(n) &= n + an\log_2 n + (n-1)b \\
&= O(n\log_2 n) \\
&= O(n\log n)
\end{aligned}$$

for any logarithmic base. It can be shown that the best, worst, and average behavior are all $O(n\log n)$ for mergesort. The only disadvantage is that a temporary array is required, of the same size as the subarray to be sorted.

So far we have seen algorithms that are $O(f(n))$ for $f(n) = \log n$, $f(n) = n$, $f(n) = n\log n$, and $f(n) = n^2$, in order of fastest increase as $n \to \infty$, as shown in Table 12.8.

12.5.9 File merge example

The merge part of mergesort is useful by itself. For example, it can be used to merge two sorted files into a single larger sorted file. The algorithm in this case is simpler than the array merge used in mergesort. No temporary storage is required. We simply read a record from each file, compare the fields that define the order of the records, and write the smaller record to the output file. This

$\log_2 n$	n	$n \log_2 n$	n^2
3.32	10	33.22	10^2
6.64	10^2	664.39	10^4
9.97	10^3	9.97×10^3	10^6
13.29	10^4	1.33×10^5	10^8
16.61	10^5	1.66×10^6	10^{10}
19.93	10^6	1.99×10^7	10^{12}
23.25	10^7	2.33×10^8	10^{14}
26.58	10^8	2.66×10^9	10^{16}
29.90	10^9	2.99×10^{10}	10^{18}

Table 12.8: Growth rates for $\log_2 n$, n, $n \log_2 n$, and n^2.

continues until one or both files are exhausted. Then if there are any records remaining in one of the input files they are written to the output file.

As an example consider files of `BankAccount` objects, using the single-line colon-separated format discussed in Chapter 12, that are sorted in order of increasing account number. For example, if the first file is `accounts1.dat`:

```
175:Linda Kerr:8008.43
424:Mary Barber:35135.5
932:Harry Garfield:4723.1
1134:Alfred Vaillancourt:51914.93
1345:Amy Flintstone:81507.31
2489:Barney Lafreniere:66568.5
7123:Jean Ebert:53361.25
7845:Marc Gardiner:7541.32
9243:Dan Sinclair:4151.34
9546:Peter Jensen:16146.29
```

and the second file is `accounts2.dat`:

```
310:Don Laing:12337.39
417:Marc Tyler:3455.42
811:Amanda Schryer:3541.15
1219:Gilles Olivier:84285.23
1765:Patricia Innes:4494.43
9623:Remi Martin:19732.12
9754:Patricia Schneider:40066.76
9912:Mike Laforge:5798.0
```

then the merged output file is

```
175:Linda Kerr:8008.43
310:Don Laing:12337.39
417:Marc Tyler:3455.42
```

```
     424:Mary Barber:35135.5
     811:Amanda Schryer:3541.15
     932:Harry Garfield:4723.1
     1134:Alfred Vaillancourt:51914.93
     1219:Gilles Olivier:84285.23
     1345:Amy Flintstone:81507.31
     1765:Patricia Innes:4494.43
     2489:Barney Lafreniere:66568.5
     7123:Jean Ebert:53361.25
     7845:Marc Gardiner:7541.32
     9243:Dan Sinclair:4151.34
     9546:Peter Jensen:16146.29
     9623:Remi Martin:19732.12
     9754:Patricia Schneider:40066.76
     9912:Mike Laforge:5798.0
```

Here is a class that does this file merge using command line arguments for the names of the two input files and the merged output file.

Class `FileMerger`

`book-project/chapter12/merge`

```
package chapter12.merge;
import custom_classes.BankAccount;
import java.io.IOException;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;

/**
 * This class uses the merge algorithm to read two bank account
 * files that are each sorted in order of increasing account number
 * and merge them into a single sorted file.
 */
public class FileMerger
{
   private File inFile1;
   private File inFile2;
   private File outFile;

   /** Create an object for given filenames.
    * @param inFile1 the first input file object
    * @param inFile2 the second input file object
    * @param outFile the output file object
    */
   public FileMerger(File inFile1, File inFile2, File outFile)
   {
      this.inFile1 = inFile1;
      this.inFile2 = inFile2;
      this.outFile = outFile;
   }

   /** Merge the two input files according to increasing account number.
    * @throws IOException
    */
   public void mergeFiles() throws IOException
   {
      BufferedAccountReader in1 =
         new BufferedAccountReader(new FileReader(inFile1));
      BufferedAccountReader in2 =
         new BufferedAccountReader(new FileReader(inFile2));
      PrintAccountWriter out =
         new PrintAccountWriter(new FileWriter(outFile));

      // Read records until end of file is reached on one or both files
      // and merge them to the output file.

      BankAccount b1 = in1.readAccount(); // get account from 1st file
      BankAccount b2 = in2.readAccount(); // get account from 2nd file

      while (b1 != null && b2 != null)
      {
         if (b1.getNumber() < b2.getNumber())
         {
            out.writeAccount(b1);   // write account from 1st file
            b1 = in1.readAccount(); // read next account from 1st file
         }
         else
         {
            out.writeAccount(b2);   // write account from 2nd file
            b2 = in2.readAccount(); // read next account from 2nd file
         }
      }

      // If there are records remaining in 1st file write them to output file.

      while (b1 != null)
      {
        out.writeAccount(b1);
        b1 = in1.readAccount();
      }

      // If there are records remaining in 2nd file write them to output file.

      while (b2 != null)
      {
         out.writeAccount(b2);
         b2 = in2.readAccount();
      }

      in1.close();
      in2.close();
      out.close();
   }

   public static void main(String[] args) throws IOException
   {
      if (args.length == 3)
      {
         File inFile1 = new File(args[0]);
         File inFile2 = new File(args[1]);
         File outFile = new File(args[2]);
         FileMerger merger = new FileMerger(inFile1, inFile2, outFile);
         merger.mergeFiles();
      }
      else
      {
         System.out.println("args: inFileName1 inFileName2 outFileName");
         System.exit(1);
      }
   }
}
```

12.5.10 Quicksort

The “divide and conquer” method used in binary search and mergesort is also the basis of the quicksort algorithm. Dividing is easy for binary search. The middle element is always used, an $\mathcal{O}(1)$ operation, and the dividing is an $\mathcal{O}(\log n)$ operation. This gives the $\mathcal{O}(\log n)$ running time. For mergesort the middle element is also used. Here the merging is an $\mathcal{O}(n)$ operation and again the dividing is $\mathcal{O}(\log n)$. This gives the $\mathcal{O}(n \log n)$ running time.

For the quicksort algorithm the dividing is $\mathcal{O}(\log n)$ but it is based on a partitioning method for dividing the array into two parts determined by choosing a pivot element. Like merge, this partitioning is an $\mathcal{O}(\log n)$ operation but the partitioning operation is not intuitive at all compared to the one defined by merge. For quicksort the two parts are not necessarily of equal size. In fact it is possible to have one part contain one element and the other part to contain the remaining elements at each stage in the recursive division process. This becomes an $\mathcal{O}(n^2)$ operation and is the reason that quicksort has an $\mathcal{O}(n \log n)$ average case behavior but only an $\mathcal{O}(n^2)$ worst case behavior.

Partitioning an array

To understand quicksort we first define a partition of the subarray $\langle a_{start}, \ldots, a_{end} \rangle$ with respect to some arbitrary element x in this subarray called the pivot element as

$$L = \langle a_{start}, \ldots, a_{split} \rangle, \quad R = \langle a_{split+1}, \ldots, a_{end} \rangle, \quad x = a_{split}$$

where the left and right subarrays are such that each element y in L satisfies $y \le a_{split}$ and each element y in R satisfies $y \ge a_{split}$. Thus, every element in L is less than or equal to the pivot element and every element in R is greater than or equal to the pivot element. We have included the pivot

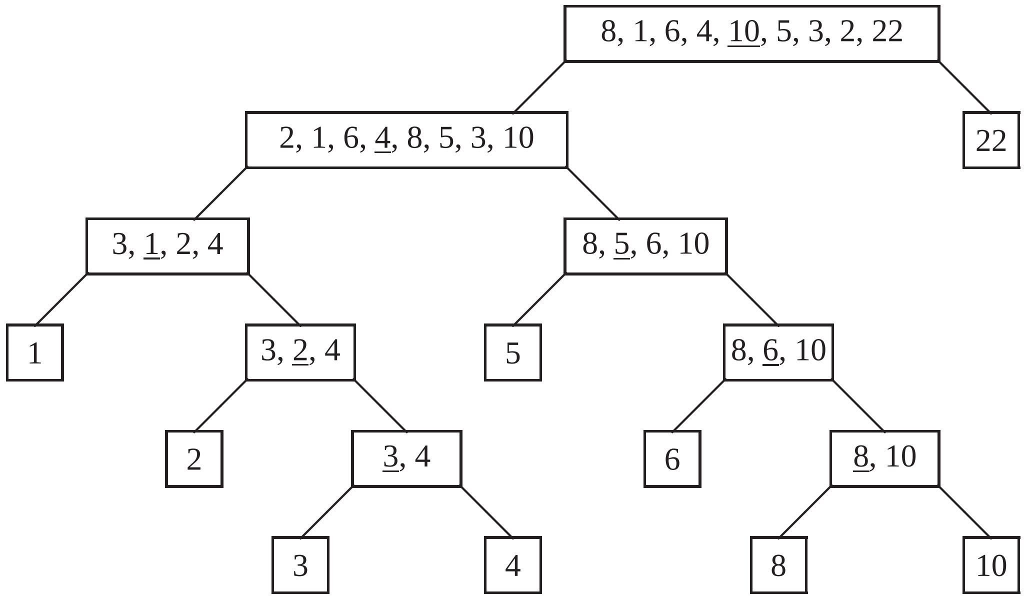


Figure 12.12: Partitioning subarrays for quicksort. The pivot for each subarray is underlined.

element in L for convenience, although it is in its correct position and will not be included in L in the recursive algorithm. Unlike mergesort this does not mean that L and R are each sorted and the pivot element does not have to be the middle element of the subarray.

For example, consider the array $\langle 8,1,6,4,10,5,3,2,22 \rangle$ and let us choose the middle element 10 as the pivot at each stage defined by the position $mid = (start + end)/2$. Then an initial partition is defined by $L = \langle 2,1,6,4,8,5,3,10 \rangle$, $R = \langle 22 \rangle$. Each element in L is less than or equal to every element in R. Since R has only one element it is already sorted. We now partition L using its middle element 4 as pivot. This gives the subarrays $\langle 3,1,2,4 \rangle$ and $\langle 8,5,6,10 \rangle$. Continuing we obtain the results shown in Figure 12.12. Since the partitioning is done in place, then 1 is in position 0, 2 is in position 1 and so on until we arrive at 22 in position 8. If we read the one-element arrays from left to right we see that the array is now sorted.

Given an algorithm for partitioning a subarray, it is easy to quicksort the array recursively: partition the subarray into the L and R subarrays and recursively quicksort L and R until any remaining subarrays have one element (base case). Because of the defining property of a partition the result will be a sorted array. Therefore the recursive quicksort method for a subarray $\langle a_{start}, \ldots, a_{end} \rangle$ has the top level recursive structure given by the Java method

```
public void quickSort(int[] a, int start, int end)
{
   if (start < end)
   {
      int split = partition(a, start, end);
      quickSort(a, start, split-1); // sort left part a[start] to a[split-1]
      quickSort(a, split+1, end); // sort right part a[split+1] to a[end]
   }
}
```

where the `partition` method returns the index, `split`, defining the two parts of the subarray. Since the pivot is in its correct position it is not included in the subarrays for either of the recursive calls to `sort`.

An implementation of partition

Designing an algorithm to perform partitioning is not easy. Essentially we need to scan the array from left to right and swap elements that are in the wrong subarray. There are several versions. Some use two index pointers, one beginning at the left and moving to the right and the other beginning at the right and moving to the left. The elements at these positions are compared and swapped if necessary. The simplest version uses only one index and can be written in Java as

```
int partition(int a[], int start, int end)
{
   // choose middle element as pivot and move it to
   // the start of the subarray temporarily.

   swap(a, (start + end)/2, start);
   int pivot = a[start];

   // partition the elements a[start+1] to a[end].
   // lastLeft is the index of the last element in the left subarray.
   // The elements a[start] to a[lastLeft] are
   // less than or equal to the pivot value.

   int lastLeft = start;
   for (int j = start+1; j <= end; j++)
   {
      if (a[j] < pivot)
      {
         lastLeft++;             // move partition index left
         swap(a, j, lastLeft);  // and swap element there with a[j]
      }
   }

   // move the pivot back to its correct position

   swap(a, start, lastLeft);
   return lastLeft;
}
```

where the `swap` method is defined as

```
void swap(int[] a, int i, int j)
{
```

`lastLeft`	loop index `j`	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	**operation**
-	-	2	1	6	4	8	5	3	10	initial array
0	1	4	1	6	2	8	5	3	10	swap 4 and 2
1	1		1							swap 1 and 1
1	2									none
2	3			2	6					swap 6 and 2
2	4									none
2	5									none
3	6				3			6		swap 6 and 3
3	7	4	1	2	3	8	5	6	10	none, loop ends
3	-	3	1	2	4	8	5	6	10	swap 4 and 3

Figure 12.13: Partitioning $\langle 2,1,6,4,8,5,3,10\rangle$ into subarrays $\langle 3,1,2,4\rangle$ and $\langle 8,5,6,10\rangle$.

```
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
```

Initially the pivot element is swapped with the first element of the subarray. Then the index pointer, `lastLeft`, is initialized. It will move to the right and at the end of the loop will be the correct position of the pivot element. The last step is to swap the pivot into this position.

As an example consider partitioning the array $\langle 2,1,6,4,8,5,3,10\rangle$, using the pivot element 4, to obtain $\langle 3,1,2,4,8,5,6,10\rangle$ which gives the partition $L = \langle 3,1,2,4\rangle$, $R = \langle 8,5,6,10\rangle$. The steps are shown in Figure 12.13. Here the first line of the table shows the initial array. The next line shows the array after the pivot element 4 in position 3 is swapped with the element, 2, in position 0. The next 7 lines correspond to the seven iterations of the for loop at the end of each iteration showing only elements that have been swapped. The last line corresponds to swapping the pivot element to its correct position given by the final value of `lastLeft` to obtain the partition $\langle 3,1,2,4\rangle$ and $\langle 8,5,6,10\rangle$.

The `quickSort` and `partition` methods are placed in a class called `IntArraySort` (see Section 12.5.13, page 696).

12.5.11 Running time for quicksort

Since the partition algorithm is $O(n)$ and the dividing should be $O(\log n)$ as in mergesort, we expect quicksort to be an $O(n\log n)$ algorithm. On average this is true, as shown below, but there are cases where the algorithm degenerates to $O(n^2)$. This occurs when the partitions at each step have one element in one part and the remaining elements in the other part. For example, since we are using the middle element as the pivot at each step, suppose the pivot is always the smallest element in the subarray. Then, since the elements of the left part are always less than or equal to the pivot, we will always have a left part with just one element (assuming there are no duplicates among the array elements). In this worst case the partitioning and the division processes are each

$O(n)$ so quicksort becomes an $O(n^2)$ algorithm.

The analysis of the average running time is the same as for mergesort. On average we expect that the partitioning of an n element subarray produces left and right parts with $n/2$ elements in each. This gives the recurrence relation

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \text{ time to partition} \\ &= 2T(n/2) + an + b \end{aligned}$$

This is the same recurrence relation obtained for mergesort so the average case behavior of quicksort is also $O(n \log n)$.

12.5.12 Simulation to compare mergesort and quicksort

We have compared the running times of the $O(n^2)$ selection and insertion sort algorithms and we now do the same for the $O(n \log n)$ algorithms, mergesort and quicksort, using the following class:

Class `FasterSortTimer`

`book-project/chapter12/sorting`

```
package chapter12.sorting;
import java.util.Scanner;
import java.util.Random;

/**
 * Estimate the average time of the merge and quick
 * sort algorithms using a given array size and number of trials
 * to average
 */
public class FasterSortTimer
{
   /**
    * Perform a sorting simulation to compare insertion
    * sort and selection sort and display results.
    */
   public void doTest()
   {
      Scanner input = new Scanner(System.in);
      String sortType = "";

      // Get the sorting algorithm

      System.out.println("Merge sort: 1");
      System.out.println("Quick sort: 2");
      System.out.println("Enter 1 or 2");
      int choice = input.nextInt(); input.nextLine();

      // Get size of array and number of trials to average

      System.out.print("Enter size of array: ");
```

```
int size = input.nextInt(); input.nextLine();
System.out.print("Enter number of trials to average: ");
int numTrials = input.nextInt(); input.nextLine();
Random rand = new Random(1234);
int[] array = new int[size];

double timeSum = 0.0;
for (int trial = 1; trial <= numTrials + 1; trial++)
{
   // fill array with random integers

   for (int k = 0; k < array.length; k++)
      array[k] = rand.nextInt();

   // compute the time for this trial

   long endTime = 0L, startTime = 0L;
   if (choice == 1)
   {
      sortType = "Merge sort";
      startTime = System.nanoTime(); // nanoseconds
      IntArraySort.mergeSort(array, 0, array.length - 1);
      endTime = System.nanoTime();
   }
   else
   {
      sortType = "Quick sort";
      startTime = System.nanoTime(); // nanoseconds
      IntArraySort.quickSort(array, 0, array.length - 1);
      endTime = System.nanoTime();
   }

   if (trial > 1) // skip first trial (do numTrials trials)
   {
      double sortTime = (endTime - startTime) / 1E9; // seconds
      System.out.println("Sort time: " + sortTime + " seconds");
      timeSum = timeSum + (double) sortTime;
   }
}

// Compute average time over all trials

double averageTime = timeSum / (double) numTrials;

// Display the results

System.out.println();
System.out.println(sortType);
System.out.println("Array size (n): " + size);
System.out.println("Number of trials: " + numTrials);
System.out.println("Average running time: " + averageTime + " seconds");
double nLogn = size * Math.log(size);
```

size n	**merge sort**	$T(n)/n\ln n$	**quicksort**	$T(n)/n\ln n$
5,000	0.034 sec	8.031×10^{-7}	0.0090 sec	2.000×10^{-7}
10,000	0.070 sec	7.589×10^{-7}	0.0192 sec	2.085×10^{-7}
20,000	0.136 sec	6.876×10^{-7}	0.0369 sec	1.863×10^{-7}
30,000	0.216 sec	6.981×10^{-7}	0.0482 sec	1.558×10^{-7}
40,000	0.294 sec	6.927×10^{-7}	0.0704 sec	1.661×10^{-7}
50,000	0.312 sec	5.760×10^{-7}	0.0860 sec	1.590×10^{-7}
100,000	0.654 sec	5.681×10^{-7}	0.1740 sec	1.508×10^{-7}
500,000	3.365 sec	5.128×10^{-7}	0.9850 sec	1.501×10^{-7}
1,000,000	7.009 sec	5.074×10^{-7}	2.0860 sec	1.510×10^{-7}
5,000,000	36.68 sec	5.145×10^{-7}	12.11 sec	1.570×10^{-7}
10,000,000	–	–	25.82 sec	1.600×10^{-7}

Table 12.9: Comparing average times for mergesort and quicksort.

```
      double ratio = averageTime / nLogn;
      System.out.println("average / n log n: " + ratio);
   }

   public static void main(String[] args)
   {
      FasterSortTimer timer = new FasterSortTimer();
      timer.doTest();
   }
}
```

Some results of running these programs on the same computer are shown in Table 12.9. For large n we have approximately

$$\begin{aligned} T(n) &= 5.1\times10^{-7}\, n\ln n, \text{ for mergesort} \\ T(n) &= 1.6\times10^{-7}\, n\ln n, \text{ for quicksort} \end{aligned}$$

showing that quicksort, on average, is more than 3 times faster than mergesort. Because mergesort requires a temporary array we ran out of memory trying it with $n = 10^7$. Either algorithm is considerably faster than selection or insertion sort as a comparison with Table 12.7 shows.

12.5.13 Class of static sorting methods

Here is the `IntArraySort` class containing the four sorting algorithms that we have developed.

Class `IntArraySort`

`book-project/chapter12/sorting`

```
package chapter12.sorting;
```

```
/**
 * Sorting methods for arrays of integers
 */
public class IntArraySort
{
   /**
    * Sort a subarray in increasing order using selection sort.
    * @param a The array
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    */
   public static void selectionSort(int[] a, int start, int end)
   {
      for (int i = start; i < end; i++)
      {
         // find position k of minimum element among
         // the elements a[i] to a[end]

         int k = i;
         for (int j = i+1; j <= end; j++)
         {
            if (a[j] < a[k])
               k = j;
         }

         // swap the smallest element found (it’s a[k]) with a[i]

         int temp = a[k];
         a[k] = a[i];
         a[i] = temp;
      }
   }

   /**
    * Sort a subarray in increasing order using insertion sort.
    * @param a The array
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    */
   public static void insertionSort(int[] a, int start, int end)
   {
      for (int i = start+1; i <= end; i++)
      {
         // Sorted part of array is a[start], ..., a[i-1]
         // Unsorted part is a[i], ..., a[end]

         int x = a[i]; // left element of unsorted part
         int j = i-1;  // right index of sorted part

         // move elements right until position for x is found.

         while ( (j >= start) && (x < a[j]) )
         {
            a[j+1] = a[j]; // move a[j] one place to the right
            j--;
         }
         a[j+1] = x; // drop x into the hole found
      }
   }

  /**
   * Sort a subarray in increasing order using merge sort.
   * @param a The array
   * @param start Index defining start of subarray
   * @param end Index defining end of subarray
   */
   public static void mergeSort(int[] a, int start, int end)
   {
      if (start == end)
         return; // one-element subarray is already sorted
      int mid = (start + end) / 2;
      mergeSort(a, start, mid); // merge sort left half
      mergeSort(a, mid+1, end); // merge sort right half
      merge(a, start, mid, end); // merge the two halves
   }

  /**
   * Merge two sorted subarrays into a sorted subarray.
   * The merge part of merge sort that takes the sorted subarrays
   * a[start] to a[split] and a[split+1] to a[end] and merges them into
   * the sorted subarray a[start] to a[end].
   * @param a The array
   * @param start Index defining start of left subarray a[start] to a[mid].
   * @param split Index defining end of left subarray
   * @param end Index defining end of right subarray a[mid+1] to a[end]
   */
   public static void merge(int[] a, int start, int split, int end)
   {
      int n = end - start + 1; // number of elements to merge
      int[] t = new int[n];    // temporary storage required
      int i = start;           // index of elements in left subarray
      int j = split + 1;       // index of elements in right subarray
      int k = 0;               // index into temporary storage

      // merge elements from left and right subarray to temp array
      // until one or both of the subarrays are exhausted

      while (i <= split && j <= end)
      {
         if (a[i] < a[j]) // element in left subarray is smaller
         {
            t[k] = a[i];     // move it to temp array t
            i++;             // index next element in left subarray
         }
         else             // element in right subarray is smaller
         {
            t[k] = a[j];     // move it to temp array t
            j++;             // index next element in right subarray
         }
         k++;             // in either case index next position in t
      }

      // copy any remaining elements from left subarray to t

      while (i <= split)
      {
         t[k] = a[i];
         i++;
         k++;
      }

      // copy any remaining elements from right subarray to t

      while (j <= end)
      {
         t[k] = a[j];
         j++;
         k++;
      }

      // copy elements to a from temporary array t. Can also use
      // System.arraycopy(t, 0, a, start, end-start+1);

      for (k = 0; k < n; k++)
         a[start+k] = t[k];
   }

   /**
    * Sort a subarray in increasing order using quicksort.
    * @param a The array
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    */
   public static void quickSort(int[] a, int start, int end)
   {
      if (start < end)
      {
         int split = partition(a, start, end);
         quickSort(a, start, split-1); // sort left part a[start] to a[split-1]
         quickSort(a, split+1, end); // sort right part a[split+1] to a[end]
      }
   }

   /**
    * Partition a subarray using the middle element as pivot.
    * This version of partition is due to Lomuto.
    * @param a The array
    * @param start Index of first subarray element
    * @param end Index of last subarray element
    * @return index <code>split</code> such that the left
    *    subarray is <code>a[start]</code> to <code>a[split]</code>,
    *    with <code>a[split]</code> being the pivot,
    *    and the right subarray is <code>a[split+1]</code>
    *    to <code>a[end]</code>.
    */
   public static int partition(int a[], int start, int end)
   {
      // choose middle element as pivot and move it to
      // the start of the subarray temporarily.

      swap(a, (start + end)/2, start);
      int pivot = a[start];

      // partition the elements a[start+1] to a[end].
      // lastLeft is the index of the last element in the left
      // subarray. The elements a[start] to a[lastLeft] are
      // less than or equal to the pivot value.

      int lastLeft = start;
      for (int j = start+1; j <= end; j++)
      {
         if (a[j] < pivot)
         {
            lastLeft++;            // move partition index right
            swap(a, j, lastLeft);  // and swap element there with a[j]
         }
      }

      swap(a, start, lastLeft); // move pivot to its correct position
      return lastLeft;
   }

   /*
    * Swap two array elements given by their indices i, j.
    *
    */
   private static void swap(int[] a, int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
}
```

12.5.14 Testing the sorting algorithms

The following `SortTester` class can be used to test the sorting algorithms.

Class `SortTester`

book-project/chapter12/sorting

```
package chapter12.sorting;
import java.util.Scanner;

/**
 * Test the sort algorithms.
 */
public class SortTester
{
   public void doTest()
   {

      Scanner input = new Scanner(System.in);

      // Read the array to sort

      System.out.print("Enter number of integers in array: ");
      int size = input.nextInt(); input.nextLine();
      int[] testArray = new int[size];
      for (int k = 0; k < testArray.length; k++)
      {
         System.out.print("Enter element " + k + ": ");
         testArray[k] = input.nextInt(); input.nextLine();
      }

      // Read the indices that define the array slice

      System.out.print("Enter start index for subarray: ");
      int start = input.nextInt(); input.nextLine();
      System.out.print("Enter end index for subarray: ");
      int end = input.nextInt(); input.nextLine();

      // sort the array using each method and display the sorted array

      int[] testArrayCopy;

      testArrayCopy = arrayCopy(testArray);
      IntArraySort.selectionSort(testArrayCopy, start, end);
      System.out.println("Selection sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);

      testArrayCopy = arrayCopy(testArray);
      IntArraySort.insertionSort(testArrayCopy, start, end);
      System.out.println("Insertion sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);

      testArrayCopy = arrayCopy(testArray);
      IntArraySort.mergeSort(testArrayCopy, start, end);
      System.out.println("Merge sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);
      testArrayCopy = arrayCopy(testArray);
      IntArraySort.quickSort(testArrayCopy, start, end);
      System.out.println("Quick sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);
   }

   private void printArray(int[] a, int start, int end)
   {
      System.out.print("[");
      for (int k = start; k <= end; k++)
      {
         System.out.print(a[k]);
         if (k < end) System.out.print(",");
      }
      System.out.print("]");
      System.out.println();
   }

   private int[] arrayCopy(int[] a)
   {
       int[] copy = new int[a.length];
       for (int k = 0; k < a.length; k++)
          copy[k] = a[k];
       return copy;
   }

   public static void main(String[] args)
   {
      new SortTester().doTest();
   }
}
```

12.6 Generic object sorting

So far our sorting algorithms in `IntArraySort` work only with arrays of type `int[]`. To sort arrays of double numbers we would have to rewrite all of them to use type `double[]`. Also we have assumed that the arrays are to be sorted in increasing order but we may want to sort in decreasing order which would require another set of classes.

The solution for versions of Java up to 1.4 is to write generic object algorithms that sort arrays of type `Object[]`. Then an array of elements of any object type can be sorted since the elements are also of type `Object`. The only problem is that we must ensure that only objects of the desired type are stored in the array and we must typecast to obtain the actual type from the object type.

In Java 5 generic parametric types were introduced so we can use a type parameter such as `E` for the element type and use an array specified as `E[]`. The advantage is that the compiler can now check that only elements of type `E` are stored in the array. When the checking is complete the type `E` can be converted by the compiler to type `Object`.

In either case (Java 1.4 or Java 5) it is necessary to have some generic object comparison method that can compare two objects to define which comes first (total ordering). For example, in the `InsertionSort` method there is the if-statement

```
if (a[j] < a[k])
   k = j;
```

The boolean expression `a[j] < a[k]` will not work if we want to compare objects. For `int` and `double` numbers there are predefined operators such as `<` that define an order. For objects we need to define the order ourselves. For example, we could sort an array of `BankAccount` objects in order of increasing bank balance, or we could do it in order of increasing account number, or even in lexicographical order using the owner name.

12.6.1 The `Comparator` interface

There is an interface called `Comparator` in package `java.util` that has a `compare` method that can be used to define a total order on objects of some type `T` in Java 5:

```
public interface Comparator<T>
{
   public int compare(T obj1, T obj2);
}
```

Prior to Java 5 this interface was given by

```
public interface Comparator
{
   public int compare(Object obj1, Object obj2);
}
```

Here `compare` returns a negative value if `obj1` is “less than” `obj2`, 0 if `obj1` is equal to `obj2`, and a positive value if `obj1` is “greater than” `obj2`. This is the same convention used by the `compareTo` method in the `String` class. Then, if `f` is an object from a class that implements this interface the generic form of the above if-statement is

```
if (f.compare(a[j], a[k]) < 0)
    k = j;
```

The opposite order can be defined simply by reversing the negative and positive return values.

Now we can rewrite all our sorting methods to use an argument array of type `Object[]` (type `E[]` in Java 5) and an extra argument to specify an object implementing the `Comparator` interface (`Comparator<E>` in Java 5).

For example, in Java 1.4 the selection sort method would now look like:

```
public static void selectionSort(Object[] a, int start, int end, Comparator f)
{
   for (int i = start; i < end; i++)
   {
      // find position k of minimum element among
```

```
         // the elements a[i] to a[end]

         int k = i;
         for (int j = i+1; j <= end; j++)
         {
            if (f.compare(a[j],a[k]) < 0)
               k = j;
         }

         // swap the smallest element found (it’s a[k]) with a[i]

         Object temp = a[k];
         a[k] = a[i];
         a[i] = temp;
      }
   }
```

and in Java 5 it would look like

```
   public static <E> void selectionSort(E[] a,int start,int end, Comparator<E> f)
   {
      for (int i = start; i < end; i++)
      {
         // find position k of minimum element among
         // the elements a[i] to a[end]

         int k = i;
         for (int j = i+1; j <= end; j++)
         {
            if (f.compare(a[j],a[k]) < 0)
               k = j;
         }

         // swap the smallest element found (it’s a[k]) with a[i]

         E temp = a[k];
         a[k] = a[i];
         a[i] = temp;
      }
   }
```

Here we specify the type before the `void` keyword (this is always done for static methods) and then we specify the array type `E[]` and use `Comparator<E>` to specify the comparator type. Also, when swapping two elements we use the type `E` for the temporary variable.

12.6.2 GenericArraySort class

We can put all our generic sorting algorithms in a static class called `GenericArraySort`. Here we show only the Java 5 version.

Class GenericArraySort

book-project/chapter12/generic

```
package chapter12.generic;
import java.util.Comparator;

/**
 * Generic sorting methods for arrays of generic type that
 * use a Comparator object to define the sort order.
 */
public class GenericArraySort
{
   /**
    * Sort a subarray in a specified order using selection sort.
    * @param a The array
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    * @param f A Comparator object defining the sort order
    */
   public static <E> void selectionSort(E[] a,
      int start, int end, Comparator<E> f)
   {
      for (int i = start; i < end; i++)
      {
         // find position k of minimum element among
         // the elements a[i] to a[end]

         int k = i;
         for (int j = i+1; j <= end; j++)
         {
            if (f.compare(a[j], a[k]) < 0)
               k = j;
         }

         // swap the smallest element found (it’s a[k]) with a[i]

         swap(a, k, i);
      }
   }

   /**
    * Sort a subarray in a specified order using insertion sort.
    * @param a The array
    * @param start Index defining start of subarray
    * @param end Index defining end of subarray
    * @param f A Comparator object defining the sort order
    */
   public static <E> void insertionSort(E[] a,
      int start, int end, Comparator<E> f)
   {
      for (int i = start+1; i <= end; i++)
      {
         // Sorted part of array is a[start], ..., a[i-1]
         // Unsorted part is a[i], ..., a[end]

         E x = a[i]; // left element of unsorted part
         int j = i-1;  // right index of sorted part

         // move elements right until position for x is found.

         while ( (j >= start) && (f.compare(x,a[j]) < 0) )
         {
            a[j+1] = a[j]; // move a[j] one place to the right
            j--;
         }
         a[j+1] = x; // drop x into the hole found
      }
   }

  /**
   * Sort a subarray in a specified order using merge sort.
   * @param a The array
   * @param start Index defining start of subarray
   * @param end Index defining end of subarray
   * @param f A Comparator object defining the sort order
   */
   public static <E> void mergeSort(E[] a,
      int start, int end, Comparator<E> f)
   {
      if (start == end)
         return; // one-element subarray is already sorted
      int mid = (start + end) / 2;
      mergeSort(a, start, mid, f); // merge sort left half
      mergeSort(a, mid+1, end, f); // merge sort right half
      merge(a, start, mid, end, f); // merge the two halves
   }

  /**
   * Merge two sorted subarrays into a sorted subarray.
   * The merge part of merge sort that takes the sorted subarrays
   * a[start] to a[split] and a[split+1] to a[end] and merges them into
   * the sorted subarray a[start] to a[end].
   * @param a The array
   * @param start Index defining start of left subarray a[start] to a[mid].
   * @param split Index defining end of left subarray
   * @param end Index defining end of right subarray a[mid+1] to a[end]
   * @param f A Comparator object defining the sort order
   */
```

```
public static <E> void merge(E[] a,
   int start, int split, int end, Comparator<E> f)
{
   int n = end - start + 1;    // number of elements to merge
   E[] t = (E[]) new Object[n]; // temporary storage required
   int i = start;              // index of elements in left subarray
   int j = split + 1;          // index of elements in right subarray
   int k = 0;                  // index into temporary storage

   // merge elements from left and right subarray to temp array
   // until one or both of the subarrays are exhausted

   while (i <= split && j <= end)
   {
      if (f.compare(a[i],a[j]) < 0) // element in left subarray is smaller
      {
         t[k] = a[i];     // move it to temp array t
         i++;             // index next element in left subarray
      }
      else             // element in right subarray is smaller
      {
         t[k] = a[j];     // move it to temp array t
         j++;             // index next element in right subarray
      }
      k++;             // in either case index next position in t
   }

   // copy any remaining elements from left subarray to t

   while (i <= split)
   {
      t[k] = a[i];
      i++;
      k++;
   }

   // copy any remaining elements from right subarray to t

   while (j <= end)
   {
      t[k] = a[j];
      j++;
      k++;
   }

   // copy elements to a from temporary array t. Can also use
   // System.arraycopy(t, 0, a, start, end-start+1);

   for (k = 0; k < n; k++)
      a[start+k] = t[k];
}
```

```
/**
 * Sort a subarray in a specified order using quicksort.
 * @param a The array
 * @param start Index defining start of subarray
 * @param end Index defining end of subarray
 * @param f A Comparator object defining the sort order
 */
public static <E> void quickSort(E[] a,
   int start, int end, Comparator<E> f)
{
   if (start < end)
   {
      int split = partition(a, start, end, f);
      quickSort(a, start, split-1, f); // sort left part a[start] to a[split-1]
      quickSort(a, split+1, end, f); // sort right part a[split+1] to a[end]
   }
}

/**
 * Partition a subarray using the middle element as pivot.
 * This version of partition is due to Lomuto.
 * @param a The array
 * @param start Index of first subarray element
 * @param end Index of last subarray element
 * @param f A Comparator object defining the sort order
 * @return index <code>split</code> such that the left
 *    subarray is <code>a[start]</code> to <code>a[split]</code>,
 *    with <code>a[split]</code> being the pivot,
 *    and the right subarray is <code>a[split+1]</code>
 *    to <code>a[end]</code>.
 */
public static <E> int partition(E a[],
   int start, int end, Comparator<E> f)
{
   // choose middle element as pivot and move it to
   // the start of the subarray temporarily.

   swap(a, (start + end)/2, start);
   E pivot = a[start];

   // partition the elements a[start+1] to a[end].
   // lastLeft is the index of the last element in the left
   // subarray. The elements a[start] to a[lastLeft] are
   // less than or equal to the pivot value.

   int lastLeft = start;
   for (int j = start+1; j <= end; j++)
   {
      if (f.compare(a[j],pivot) < 0)
      {
         lastLeft++;            // move partition index right
         swap(a, j, lastLeft);  // and swap element there with a[j]
```

```
            }
         }

         swap(a, start, lastLeft); // move pivot to its correct position
         return lastLeft;
      }

      /*
       * Swap two array elements given by their indices i, j.
       *
       */
      private static <E> void swap(E[] a, int i, int j)
      {
         E temp = a[i];
         a[i] = a[j];
         a[j] = temp;
      }
   }
```

12.6.3 Sorting strings in lexicographical order

As an example, suppose you want to sort an array of strings in increasing lexicographical order. First define this order in the following simple class that implements the `Comparator` interface:

Class `StringComparator`

`book-project/chapter12/generic`

```
package chapter12.generic;
import java.util.Comparator;

/**
 * An implementation of the Comparator interface for strings
 * in lexicographical order.
 */
public class StringComparator implements Comparator<String>
{
   /**
    * Compare two strings lexicographically
    * @param s1 first string
    * @param s2 second string
    * @return -1 if s1 precedes s2, 0 if s1 is equal
    * to s2, and 1 if s1 follows s2
    */
   public int compare(String s1, String s2)
   {
      return s1.compareTo(s2);
   }
}
```

Note that we have used `Comparator<String>` to indicate that the comparator object is comparing two strings and we have used the `compareTo` method in the `String` class to do the comparison.

Then the following statement can be used to sort a subarray of a string array called `s` using selection sort.

```
GenericArraySort.selectionSort(s, start, end, new StringComparator());
```

The entire string array can be sorted using

```
GenericArraySort.selectionSort(s, 0, s.length - 1, new StringComparator());
```

To sort an array of strings in decreasing lexicographical order we could use the following class:

Class `StringDecreasingComparator`

`book-project/chapter12/generic`

```
package chapter12.generic;
import java.util.Comparator;

/**
 * An implementation of the Comparator interface for strings
 * in reverse lexicographical order.
 */
public class StringDecreasingComparator implements Comparator<String>
{
   /**
    * Compare two strings in reverse lexicographical order
    * @param s1 first string
    * @param s2 second string
    * @return -1 if s2 precedes s1, 0 if s1 is equal
    * to s2, and 1 if s2 follows s1
    */
   public int compare(String s1, String s2)
   {
      return s2.compareTo(s1);
   }
}
```

Here is a complete program that sorts an array of strings using all of the sorting methods.

Class `GenericStringSortTester`

`book-project/chapter12/generic`

```
package chapter12.generic;
import java.util.Comparator;
import java.util.Scanner;

/**
 * Using the generic version of the sorting algorithms to sort
 * an array of strings in increasing and decreasing
 * lexicographical order.
 */
```

```
public class GenericStringSortTester
{
   public void doTest(Comparator<String> comp)
   {
      Scanner input = new Scanner(System.in);

      // Read the array

      System.out.print("Enter number of strings in array: ");
      int size = input.nextInt(); input.nextLine();
      String[] testArray = new String[size];
      for (int k = 0; k < testArray.length; k++)
      {
         System.out.print("Enter string element " + k + ": ");
         testArray[k] = input.nextLine();
      }

      // Read the indices that define the array slice

      System.out.print("Enter start index for subarray: ");
      int start = input.nextInt(); input.nextLine();
      System.out.print("Enter end index for subarray: ");
      int end = input.nextInt(); input.nextLine();

      // sort the array using each method and display the sorted array

      String[] testArrayCopy;

      testArrayCopy = arrayCopy(testArray);
      GenericArraySort.selectionSort(testArrayCopy, start, end, comp);
      System.out.println("Selection sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);

      testArrayCopy = arrayCopy(testArray);
      GenericArraySort.insertionSort(testArrayCopy, start, end, comp);
      System.out.println("Insertion Sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);

      testArrayCopy = arrayCopy(testArray);
      GenericArraySort.mergeSort(testArrayCopy, start, end, comp);
      System.out.println("Merge sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);

      testArrayCopy = arrayCopy(testArray);
      GenericArraySort.quickSort(testArrayCopy, start, end, comp);
      System.out.println("Quick sort: Sorted subarray is");
      printArray(testArrayCopy, start, end);
   }

   private <E> void printArray(E[] a, int start, int end)
   {
      System.out.print("[");
      for (int k = start; k <= end; k++)
      {
         System.out.print(a[k]);
         if (k < end) System.out.print(",");
      }
      System.out.print("]");
      System.out.println();
   }

   private String[] arrayCopy(String[] a)
   {
       String[] copy = new String[a.length];
       for (int k = 0; k < a.length; k++)
          copy[k] = a[k];
       return copy;
   }

   public static void main(String[] args)
   {
      System.out.println("Output for normal sort order");
      new GenericStringSortTester().doTest(new StringComparator());
      System.out.println("Output for reverse sort order");
      new GenericStringSortTester().doTest(new StringDecreasingComparator());
   }
}
```

Some output is

```
Output for normal sort order
Enter number of strings in array: 4
Enter string element 0: one
Enter string element 1: two
Enter string element 2: three
Enter string element 3: four
Enter start index for subarray: 0
Enter end index for subarray: 3
Selection sort: Sorted subarray is
[four,one,three,two]
Insertion Sort: Sorted subarray is
[four,one,three,two]
Merge sort: Sorted subarray is
[four,one,three,two]
Quick sort: Sorted subarray is
[four,one,three,two]
Output for reverse sort order
Enter number of strings in array: 4
Enter string element 0: one
Enter string element 1: two
Enter string element 2: three
Enter string element 3: four
Enter start index for subarray: 0
Enter end index for subarray: 3
Selection sort: Sorted subarray is
[two,three,one,four]
Insertion Sort: Sorted subarray is
[two,three,one,four]
Merge sort: Sorted subarray is
[two,three,one,four]
Quick sort: Sorted subarray is
[two,three,one,four]
```

12.6.4 Comparing `BankAccount` objects

As another example, if you want to sort an array of `BankAccount` objects in order of increasing account number, the following class defines the order.

Class `AccountNumberComparator`

`book-project/chapter12/generic`

```
package chapter12.generic;
import custom_classes.BankAccount;
import java.util.Comparator;

public class AccountNumberComparator implements Comparator<BankAccount>
{
   public int compare(BankAccount b1, BankAccount b2)
   {
      return b1.getNumber() - b2.getNumber();
   }
}
```

Here we use a subtraction so that a negative number is returned in case `b1` has a smaller account number than `b2`.

Similarly, if you want to sort in order of increasing balance use the class

Class `AccountBalanceComparator`

`book-project/chapter12/generic`

```
package chapter12.generic;
import custom_classes.BankAccount;
import java.util.Comparator;

public class AccountBalanceComparator implements Comparator<BankAccount>
{
   public int compare(BankAccount b1, BankAccount b2)
   {
      double diff = b1.getBalance() - b2.getBalance();
      if (diff < 0.0) return -1;
      else if (diff == 0.0) return 0;
      else return 1;
   }
}
```

Here is class to test these sort orders.

Class `BankAccountSortTester`

`book-project/chapter12/generic`

```
package chapter12.generic;
import custom_classes.BankAccount;
import custom_classes.JointBankAccount;

public class BankAccountSortTester
{
   public void doTest()
   {
      BankAccount[] b = new BankAccount[3];
      b[0] = new BankAccount(321, "Fred", 150.0);
      b[1] = new BankAccount(234, "Gord", 350.0);
      b[2] = new JointBankAccount(123,
            "Jack", "Jill", 200.0);

      GenericArraySort.selectionSort(b, 0, 2,
         new AccountNumberComparator());
      System.out.println("Order: number");
      for (int k = 0; k < 3; k++)
         System.out.println(b[k]);

      System.out.println("Order: balance");
      GenericArraySort.selectionSort(b, 0, 2,
            new AccountBalanceComparator());
         for (int k = 0; k < 3; k++)
            System.out.println(b[k]);
   }
   public static void main(String[] args)
   {
      new BankAccountSortTester().doTest();
   }
}
```

12.7 The `Arrays` Class For Searching and Sorting

There is a class called `Arrays` in package `java.util` that has many static methods for searching and sorting arrays. Normally you should use these methods. However, it is important to know how to develop, test, compare and determine the efficiency of searching and sorting algorithms so the algorithms developed in this chapter have pedagogical value.

The generic versions of these searching and sorting methods require that a total order be defined on the array elements. This can be done in two ways:

- The array elements belong to some class that implements the `Comparable` interface. The order defined by this interface is called the natural order. The `String` class implements `Comaparable`.
- If the class does not implement the `Comparable` interface or a different order is desired then we use a class that implements the `Comparator` interface and use an object from this class as a method argument to specify the order.

We have considered the `Comparator` interface in the `GenericArraySort` class on page 705.

12.7.1 `Comparable` interface

The `Comparable` interface resides in package `java.lang` (not `java.util` like `Comparator`):

```
public interface Comparable
{
   public int compareTo(Object obj)
}
```

In Java 5 this interface is defined as

```
public interface Comparable<E>
{
   public int compareTo(E element)
}
```

This interface, like `Comparator`, defines a total ordering on the objects of some class. However it is used in a different way. To use `Comparable` to order the objects of some class it is necessary that this class implement the `Comparable` interface (`String` for example).

12.7.2 Searching algorithms in the `java.util.Arrays` class

The `Arrays` class in package `java.util` contains the following searching algorithms:

- **`static int binarySearch(double[] a, double key)`**

  Search the `double` array **`a`** for the value **`key`** and return the index at which the value is found. If the value is not found then return the negative value $-k-1$ where k is the position at which **`key`** would need to be inserted to keep the array in sorted order

  There are similar methods for arrays of all primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`)

- **`static int binarySearch(Object[] a, double key)`**

  This is the generic object version with array type **`Object[]`**. The actual array element type must implement the `Comparable` interface.

- **`static <E> int binarySearch(E[] a, E key, Comparator<? super E> c)`**

  This is the Java 5 version that uses a generic method with an array of type **`E`** and a comparator for **`E`** or any superclass of **`E`**. If the comparator argument **`c`** is `null` then the natural sort order provided by **`E`** is used, assuming that **`E`** implements the `Comparable<E>` interface.

■ EXAMPLE 12.5 **(Searching using `Arrays.binarySearch`)** Using a `null` object for the comparator the statements

```
String[] names = {"Bill", "Fred", "Gord", "Harry"};
System.out.println(Arrays.toString(names));
int k = Arrays.binarySearch(names, "Fred", null);
System.out.println("Index for Fred is " + k);
k = Arrays.binarySearch(names, "Bob", null);
System.out.println("Index for Bob is " + k);
```

produce the output

```
[Bill, Fred, Gord, Harry]
Index for Fred is 1
Index for Bob is -2
```

which shows that Fred was found at index 1. Bob was not found but the correct sorted location would be k where $-k-1=-2$, so $k=-1+2=1$ (at position 1). ■

12.7.3 Sorting algorithms in the `java.util.Arrays` class

The `Arrays` class in package `java.util` also contains the following sorting algorithms:

- **`static void sort(double[] a)`**

  Sort the `double` array `a` in increasing numerical order. There are similar methods for arrays of all primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`)

- **`static void sort(double[] a, int fromIndex, int toIndex)`**

  Sort the `double` array **`a`** in increasing numerical order using the subarray beginning at index **`fromIndex`** and ending at index **`toIndex - 1`**, **not `toIndex`**. There are similar methods for arrays of all primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`)

- **`static void sort(Object[] a)`**
  `static void sort(Object[] a, int fromIndex, int toIndex)`

  These are the generic object version with array type **`Object[]`**. The actual element type must implement the `Comparable` interface.

- **`static <E> void sort(E[] a, Comparator<? super E> c)`**
  `static <E> void sort(E[] a, int fromIndex, int toIndex,`
  `   Comparator<? super E> c)`

  These are the Java 5 versions that uses a generic method with an array of type **`E`** and a comparator for any type **`E`** or any superclass of **`E`**.

■ EXAMPLE 12.6 **(Sorting using `Arrays.sort`)** Using a `StringDecreasingComparator` (see page 710) and the natural order the statements

```
String[] names = {"Harry", "Gord", "Fred", "Bill"};
System.out.println(Arrays.toString(names));
Arrays.sort(names, null);
System.out.println(Arrays.toString(names));
Arrays.sort(names, new StringDecreasingComparator());
System.out.println(Arrays.toString(names));
```

produce the output

```
[Harry, Gord, Fred, Bill]
[Bill, Fred, Gord, Harry]
[Harry, Gord, Fred, Bill]
```

which shows that the original string array (ordered in decreasing order) has been sorted in the natural increasing order and then this string array has been sorted in decreasing order to give back the original string array. ■

12.8 Exercises

► **Exercise 12.1 (Recursive version of `findMinimum`)**
Write a recursive version of the `findMinimum` method that has the prototype

```
int findMinimum(int[] a, int start, int end)
```

and returns the first index at which the minimum occurs

► **Exercise 12.2 (Another recursive version of `findMinimum`)**
Write a recursive version of the `findMinimum` method that returns the minimum rather than the index.

► **Exercise 12.3 (Searching for all occurrences of a given element)**
Write a pseudo-code algorithm that finds all occurrences of a given element in a subarray of a given array. The algorithm should return an array containing the indices at which the element occurs. Write a Java method with prototype

```
int[] findAllElements(int[] a, int x, int start, int end)
```

that implements this algorithm, where `a` is the array, `x` is the element to search for, and `start` and `end` define the subarray. For example, if the array is $\langle 4,3,8,65,32,65,17,65\rangle$, the subarray is the entire array, and the element to search for is 65, then the array of indices returned is $\langle 3,5,7\rangle$.

► Exercise 12.4 (Recursive binary search for String arrays)
Write a recursive binary search method with prototype

```
static int binarySearch(String[] s, String target, int start, int end)
```

that searches the subarray `s[start]` to `s[end]` of a `String` array `s` for a given `target` string. Assume that the string array is sorted in increasing lexicographical order.

► Exercise 12.5 (Modified binary search algorithm)
Modify the recursive and non-recursive binary search algorithms so they work like the ones in the `Arrays` class in case the element is not found. This means that instead of returning -1 when the element is not found the algorithm should return $-k-1$ where k is the array position at which the element could be inserted to keep the array sorted. The value $-k-1$ is chosen so that the result returned is always negative when an element is not found (the minimum value of k would be 0) and always non-negative when the element is found.

Implement this algorithm as a Java method for searching an array of strings with prototype

```
static int binarySearch(String[] a, String key)
```

Implement this algorithm as a Java 5 method for searching an `ArrayList<String>` of strings with prototype

```
static int binarySearch(ArrayList<String> a, String key)
```

► Exercise 12.6 (Insertion into an ordered list)
Use the Java 5 version of binary search from Exercise 12.5 to write a method with prototype

```
static int insertInOrderedList(ArrayList<String> a, String key)
```

If `key` is not found then it should be inserted into the list in the correct position to keep the list sorted. The return value is the position of `key` in the list. Hint: the `ArrayList<T>` class has a method with prototype

```
public void add(int k, T element)
```

that inserts an `element` of type `T` into the list at index `k`.

► Exercise 12.7 (Reversing the sort order)
Suppose you have a subarray of type `int[]` that is already sorted in increasing order. Write an $O(n)$ algorithm that will sort it in decreasing order. Write a Java method called `reverseSortOrder` with prototype

```
void reverseSortOrder(int[] a, int start, int end)
```

for this algorithm.

► **Exercise 12.8 (Comparing running times on your computer)**
Reproduce the running time results in Table 12.7 and Table 12.9 using your computer.

► **Exercise 12.9 (A version of `GenericArraySort` for `ArrayList<E>` objects)**
Write a version of the `GenericArraySort` class called `GenericArrayListSort` with methods that sort an `ArrayList<E>` object instead of an array object of type `E[]`.

► **Exercise 12.10 (A `GenericArraySearch` class)**
Using the `GenericArraySort` class as a guide, write a class called `GenericArraySearch` that implements generic versions of the linear, recursive binary, and non-recursive binary search algorithms that were written for arrays of type `int[]`.

► **Exercise 12.11 (A version of `GenericArraySearch` for `ArrayList<E>` objects)**
Write a version of the `GenericArraySearch` class called `GenericArrayListSearch` with methods that search an `ArrayList<E>` object instead of an array object of type `E[]`.

► **Exercise 12.12 (Sorting a file of `BankAccount` objects)**
Using the `GenericArrayListSort` class write a class called `AccountSorter` that reads a file of `BankAccount` objects in the single-line colon-separated format into an `ArrayList` object and writes a new file that is sorted in order of increasing account number using the quicksort method in the `GenericArrayListSort` class from Exercise 12.9.

► **Exercise 12.13 (Sorting a phone book file)**
Using the `GenericArraySort` class write a program class called `PhoneBookSorter` that reads a file of `PhoneBookEntry` objects (see Chapter 11 exercises) in the single-line colon-separated format into an `ArrayList<PhoneBookEntry>` object and writes a new file that is sorted in lexicographical order on the name using the quicksort method in the `GenericArrayListSort` class from Exercise 12.9.

► **Exercise 12.14 (Using `GenericArraySort` to sort arrays of primitive type)**
How can you use `GenericArraySort` to sort an array of type `int[]` or of type `double[]`, noting that `int` and `double` are primitive types, not `Object` types? Write a Java program class called `DoubleSortTester` that reads an array of type `double[]` and sorts it using one of the sort methods in `GenericArraySort` HINT: Use the `Double` wrapper class.

► **Exercise 12.15 (Sorting arrays of rectangles)**
Write a `Rectangle` class that has private data fields for the height and width of the rectangle, a constructor for a rectangle, given its width and height, get methods for the width and height, a `toString` method, and a method to return the area of the rectangle.

The class should implement the `Comparable<Rectangle>` interface using the area to define the order: one rectangle is less than another if it has a smaller area. Two rectangles are equal if they have the same area.

Also write a `RectangleComparator` class that implements the `Comparator<Rectangle>` interface to define the following ordering of rectangles. One rectangle is less than another if its width is smaller. In case the widths are the same then use the height and define one rectangle to be less than another of the same width if the height is smaller. In case both the height and the width are the same then the rectangles are equal.

Write a tester class called `RectangleSortTester` that constructs an array of `Rectangle` objects and uses the sorting methods in the `Arrays` class to sort the array using the order defined by the `Comparable<Rectangle>` interface. Use the order defined by `RectangleComparator`.

► Exercise 12.16 (Comparing two versions of partition)
Here is a different version of the `partition` method for arrays of type `int[]` than the one we have used in quicksort.

```
public static int partition(int a[], int start, int end)
{
   int left = start;
   int right = end;

   // choose middle element as pivot and move it to
   // the end of the array temporarily

   swap(a, (start + end)/2, end);
   int pivot = a[end];

   while (left < right)
   {
      // search left part for element larger than pivot
      while (left < right && a[left] <= pivot) left++;

      // search right part for element smaller than pivot
      while (left < right && a[right] >= pivot) right--;

      // if we find a pair of elements in the wrong parts
      // swap them and look for more

      if (left < right)
      {
         swap(a, left, right);
         left++; // look for more
      }
   }
   swap(a, left, end); // put pivot back in correct position
   return left;
}
```

Test the running time of the two versions to see if one is faster than the other.

► Exercise 12.17 (Graphical simulation of sorting algorithms)
Write a graphical simulation for one of the sorting algorithms. First choose an initial random array of *n* integers. Next scale the values so that the largest one is the height of the highest vertical bar

that will fit. Then each integer in the array can be drawn as a vertical bar using the array index as a horizontal coordinate. Now apply a sorting algorithm and after each iteration update the bar graph by calling the `repaint` method. When the sort finishes the bar heights should increase from left to right.

Chapter 13

Introduction to Data Types and Structures

Abstract Data Types and the Java Collections Framework

Outline

Abstract data types

Implementing an ADT

Java Collections Framework (JCF)

`Collection<E>` and `Set<E>` interfaces

Set implementations and examples

`List<E>` and `ListIterator<E>` interfaces

List implementations and examples

Map data type

`Map<K,V>` interface

Map implementations and examples

Recursion examples using maps

`Collections` utility class

Sorting examples

13.1 Introduction

In this chapter we consider abstract data types and their implementations. Simple examples include a fixed size bag ADT, a dynamic size bag ADT and a dynamic size array ADT. In each case simple versions of these ADTs are designed using Java interfaces and implemented using array data structures.

Next we give an overview of some of the important ADTs such as sets, lists and maps that are part of the Java Collections Framework (JCF). Here we concentrate on using the ADTs and not on how they are implemented, which is left for a course on data structures.

13.2 Abstract data types

A **data type** is a set of values (the data) and a set of operations defined on the data. An **implementation** of a data type is an expression of the data and operations in terms of a specific programming language such as Java or C++. An **abstract data type** (ADT) is a specification of a data type in a formal sense without regard to any particular implementation or programming language. Finally, a **realization** of an ADT involves two parts

- the interface, specification, or documentation of the ADT: what is the purpose of each operation and what is the syntax for using it.
- the implementation of the ADT: how is each operation expressed using the **data structures** and statements of a programming language.

The ADT itself is concerned only with the specification or interface details, not the implementation details. This separation is important. In order to use an ADT the client or user needs to know only what the operations do, not how they do it. Ideally this means that the implementation can be changed, to be more efficient for example, and the user does not need to modify programs that use the ADT since the interface has not changed.

With object-oriented programming languages such as Java and C++ there is a natural correspondence between a data type and a class. The class defines the set of operations that are permissible: they are the public methods of the class. The data is represented by the instance data fields. Each object (instance of the class) encapsulates a particular state: set of values of the data fields.

In Java the separation of specification and implementation details can easily be obtained using the Javadoc program which produces the specification (public interface) for each class. The user can simply read this documenation to find out how to use the class. It is also possible to use a Java interface for the specification of an ADT since this interface contains no implementation details, only method prototypes: any class that implements the interface provides a particular implementation of the ADT.

13.2.1 Classification of ADT operations

The various operations (methods) that are defined by an ADT can be grouped into several categories, depending on how they affect the data of an object:

Create operation

It is always necessary to create an object before it can be used. In Java this is done using the class constructors.

Copy operation

The availability of this operation depends on the particular ADT. In many cases it is not needed or desired. If present, the meaning (semantics) of the operation also depends on the particular ADT. In some cases copy means make a true copy of the object and all its data fields, and all their data fields, and so on, and in other cases it may mean to simply make a new reference to an object. In other words, the reference to the object is being copied, not the object itself. In this case there is only one object and it is shared among all the references to it. This makes sense for objects that occupy large amounts of memory and in many other cases as well. Both types of operation can even be included in the same ADT. In some languages the copy operation can have explicit and implicit versions. In Java the implicit operation, defined by assignment or method argument passing, always copies references but it is possible to make other kinds of explicit copies using a copy constructor or by overriding the `clone` method inherited from the `Object` class.

Destroy operation

Since objects take up space in memory it is necessary to reclaim this space when an object is no longer needed. This operation is often called the *destroy* operation. In Java there is no explicit destroy operation since the built-in garbage collector takes on this responsibility: when there are no more references to an object it is eventually garbage-collected.

Modification operations

Every object of an ADT encapsulates data and for some ADTs we need operations that can modify this data. These operations act on objects and change one or more of their data fields. Sometimes they are called **mutator** operations. If an ADT has no mutator operations then the state cannot be changed after an object has been created and the ADT is said to be **immutable**, otherwise it is **mutable**.

Inquiry operations

An inquiry operation inspects or retrieves the value of a data field without modification. It is possible to completely hide all or part of the internal state of an object simply by not providing the corresponding inquiry operations.

13.2.2 Pre- and post-conditions

To document the operations of an ADT pre-conditions and post-conditions can be used.

Pre-conditions They are the conditions that must be true before an operation is executed in order that the operation is guaranteed to complete successfully. These conditions can be expressed in terms of the state of the object before the operation is applied to the object. A pre-condition may or may not be needed.

Post-conditions They are the conditions that will be true after an operation completes successfully. These conditions can be expressed in terms of the state of the object after the operation has been applied to the object.

Together the pre- and post-conditions form a contract between the implementer of the method and the user of the method.

13.2.3 Simple ADT examples

The simplest examples of ADTs are the numeric, character, and boolean types. Most programming languages have realizations of them as fundamental types which are used to build more complex structured ADTs. Some typical types in these categories are

An integer ADT

Mathematically the data values here can be chosen as all integers n such that $-\infty \le n \le \infty$. Another possibility is to consider only non-negative integers n satisfying $0 \le n \le \infty$.

A typical set of operations might be the standard arithmetic operations *add*, *subtract*, *multiply*, *integer quotient* and *integer remainder*, boolean valued operations such as *equal*, *notEqual*, and the relational operators $<$, $\le$, $>$, $\ge$. An assignment operation would also be needed.

These are infinite data types since there are an infinite number of integers. Therefore any realization would need to restrict the data values to a finite subset. Some common possibilities are 8-bit, 16-bit, 32-bit, or 64-bit representations which may be signed or unsigned (non-negative values).

For example, in Java there is an 8-bit `byte` type with range $-2^7 \le n \le 2^7 - 1$, a 16-bit `short` type with range $-2^{15} \le n \le 2^{15} - 1$, a 32-bit `int` type with range $-2^{31} \le n \le 2^{31} - 1$, and a 64-bit `long` type with range $-2^{63} \le n \le 2^{63} - 1$.

A floating point ADT

Here the data values are floating point numbers. In scientific notation a floating point number would have the form $x = m \times 10^e$ where m is the mantissa and e is the exponent.

A typical set of operations would be similar to those for integers except the divide operation is now a floating point division. An assignment operation would also be needed.

For example, in Java there is a single precision 32-bit `float` type and a double precision 64-bit `double` type. The standard IEEE representation is complicated but necessary to ensure that floating point arithmetic is portable. Most processors support this standard. A single precision number x is either 0, $-3.40 \times 10^{38} \le x \le -1.40 \times 10^{-45}$ or $1.40 \times 10^{-45} \le x \le 3.40 \times 10^{38}$. A double precision number x is either 0, $-1.80 \times 10^{308} \le x \le -4.94 \times 10^{-324}$ or $4.94 \times 10^{-324} \le x \le 1.80 \times 10^{308}$.

A character ADT

Here the data is the set of characters from some character set such as ASCII or Unicode. Internally each character is represented by an unsigned integer n in the range $0 \le n \le N$ for some N.

A typical set of operations might include operations to convert from upper case to lower case and vice versa, operations to compare two characters to see if they are equal or to see if one precedes another in the lexicographical ordering defined on the characters, or an assignment operation.

For example, in Java the `char` type is an unsigned 16-bit integer type with Unicode character code n satisfying $0 \le n \le 65535$.

A boolean ADT

Here there are only two data values which can be denoted by false and true. Other possibilities are to use 0 for false and 1 for true, or 0 for false and any non-zero number for true.

A typical set of operations would be an assignment operation, an operation to test for false and one to test for true.

13.2.4 Some common structured ADTs

A structured ADT is one that is defined in terms of another ADT using to some data structure. For example, an array of integers would be defined in terms of an integer ADT and a string ADT would be defined in terms of a character ADT. These two structured ADTs are the most common and are available in most programming languages.

The array ADT

An array consists of n elements $[a_0, a_1, \ldots, a_{n-1}]$. Here the data consists of these arrays and each array element a_k belongs to some other ADT. The subscript k is called the array index. The starting index may be 0, 1, or user defined. In C++ and Java array indices begin at index 0.

The basic array operations are to *get* the value of the k-th element and *set* a new value for the k-th element. In C++ and Java the *get* operation is denoted by `x = a[k]` and the *set* operation is denoted by `a[k] = x`. This also means that an array is a mutable ADT.

The standard array ADT is of fixed size: once created its size cannot be changed. The standard arrays in C++ and Java are of this type. However we will see that it is easy to create a dynamic array ADT (resizable) which can be expanded in size if needed to accommodate more elements.

The string ADT

Strings are like arrays of characters but the operations can be quite different. Both mutable and immutable string ADTs are common. For example, in Java the `String` class represents an immutable fixed size ADT and the `StringBuilder` class represents a dynamic mutable ADT.

Some immutable string operations are to *get* the k-th character, construct a substring, construct upper case or lower case versions, and compare two strings using the lexicographical order defined on the underlying character set.

Some mutable operations are to *set* the *k*-th character to a new value, and *append* a character or string to the end of a string.

13.2.5 User defined ADT examples

We are not limited to the standard ADTs that have implementations already available in a computer language or a system defined library of ADTs. We can write our own specifications for an ADT and implement it in any language. Here we give two examples. We will show how to implement them in Java.

A dynamic array ADT

Here the data elements are arrays $[a_0, a_1, \ldots, a_{n-1}]$. This is a mutable ADT and the basic operations would be *get*, to get the *k*-th array element, and *set*, to set a new value for the *k*-th array element. Also the array size can be increased automatically as needed (doubled in size when full, for example) or by applying some expand operation that increases the array size by a specified amount.

A bag ADT

Here the data elements are bags. Each bag is a container that holds a collection of elements of some type. There is no defined order on the bag elements as there are for arrays. In mathematics a bag is often called a multi-set (no order, but duplicate elements are allowed) in contrast to sets for which there can be no duplicates.

Bags are usually designed to be mutable and dynamic so a basic set of operations are *add*, to add another element to a bag, *remove*, to remove a specified element from a bag, and *contains* which tests if a specified element is in a bag.

13.3 Implementing an ADT

We now show how to implement the bag and dynamic array ADTs. The first step is to write a specification or design of the data type, indicating what each operation does. This could be done with a Java interface followed by the design of the class implementing the interface, indicating each constructor and method body by $\{\ldots\}$.

Whether an interface is being used or not the class design should always include constructor prototypes since they are never included in an interface.

Once the design is finished it is possible to write some statements that use the ADT to 'try out' the syntax of the operations as given by the instance method prototypes. Finally, the implementation must be written (data fields, constructor and method bodies). This involves choosing some data structure to represent the data encapsulated by the objects.

In Java all data types except for the eight primitive ones (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`) are expressed as objects from some class. This presents a problem in the design of a generic type since generic types must be object types (reference types) and we cannot directly use the `int` type as a generic type. To allow primitive types to be used as objects there are wrapper classes in Java for each primitive type. For example the `Integer` class can be used as an object version of the `int` type. In Java 5 auto boxing and unboxing make this easy.

Finally, when the implementation is complete, its operations must be tested.

13.3.1 Implementation of the `Bag<E>` ADT

First we write a fixed size implementation of the bag ADT called `FixedBag<E>` using the generic type `E` for the elements in the bag. This means that once constructed for a given maximum size (number of elements) this size cannot be changed. Then we will make a simple modification to obtain a dynamic version called `DynamicBag<E>`.

Designing the `Bag<E>` ADT

Here we illustrate the use of an interface to specify the design of an ADT. Both the fixed size and dynamic versions of the ADT will implement the following interface.

Interface `Bag<E>`

`book-project/chapter13/bags`

```
package chapter13.bags;
/**
 * A simple mutable generic bag ADT.
 * @param <E> type of elements in the bag
 */
public interface Bag<E>
{
   /**
    * Return current number of elements in this bag.
    * @return current number of elements in this bag
    */
   int size();

   /**
    * Return true if this bag is empty else false.
    * @return true if this bag is empty else false
    */
   boolean isEmpty();

   /**
    * Add another element to this bag if there is room.
    * @param element the element to add
    * @return true if add was successful else false.
    */
   boolean add(E element);

   /**
    * Remove a given element from this bag.
    * @param element the element to remove
    * @return true if the element was removed.
    * A false return value occurs if element was
    * not in this bag.
    */
   boolean remove(E element);

   /**
    * Check if a given element is in this bag.
    * @param element the element to check
    * @return true if element is in this bag else false
    */
   boolean contains(E element);
}
```

We have not included the `public` modifier on the method prototypes in the interface. It is redundant since all methods in an interface are public.

Designing a fixed size implementation

The fixed size bag implementation has the form

```
public class FixedBag<E> implements Bag<E>
{
   // instance data fields will go here

   public FixedBag(int bagSize) {...}
   public FixedBag() {...}
   public FixedBag(FixedBag<E> b) {...}

   public int size() {...}
   public boolean isEmpty() {...}
   public boolean add(E element) {...}
   public boolean remove(E element) {...}
   public boolean contains(E element) {...}

   public String toString() {...}
}
```

Javadoc comments have been omitted. They are shown later in the final version of the class. Here we have three constructors. The first specifies the maximum number of elements that can be added to the bag and the no-arg constructor gives a bag with a maximum size of 10 elements. The third constructor is called a **copy constructor**. Its purpose is to construct a copy of the bag given by the argument `b`.

The `toString` method is used to return a string representation of the elements in the bag. We didn't need to include the `toString` prototype in the `Bag<E>` interface since every class inherits a `toString` method.

Also, for this fixed size implementation the `add` method would return false if the bag is already full.

According to this design we can construct a bag containing a maximum of 5 integers and add the integers 1, 2, and 3 to it using the statements

```
Bag<Integer> b = new FixedBag<Integer>(5);
b.add(1); b.add(2); b.add(3);
System.out.println(b);
```

Autoboxing is being used here: the compiler understands that `b.add(1)` means to replace `1` by the wrapper class object `new Integer(1)` and use `b.add(new Integer(1))`.

It is important to use the interface type on the left side of the constructor statement. This makes it easier to switch to another implementation class, such as a dynamic one in this case. This is sometimes called "programming to an interface".

Our bag design is minimal. For example it is not possible with this design to take a bag of integers and remove all even integers or display the bag elements one per line. This would require an iterator and will be discussed later.

■ EXAMPLE 13.1 **(Filling a fixed size bag)** The statements

```
Bag<Integer> bag = new FixedBag<Integer>(10);
for (int k = 1; k <= 10; k++)
   bag.add(k);
```

construct a fixed bag of size 10 and fill it with the numbers 1 to 10. ■

■ EXAMPLE 13.2 **(Filling a fixed size bag)** The statements

```
Bag<Integer> bag = new FixedBag<Integer>(10);
int k = 1;
while (bag.add(k))
   k++;
```

construct a fixed bag of size 10 and fill it with the numbers 1 to 10 using the `add` method to detect when the bag is full. ■

Choosing a data structure

The next step is to choose a data structure to hold the bag elements. Here we choose a fixed size array called `data` such that if the number of elements currently in the bag is `size` then these elements are stored in `data[0]`, `data[1]`, ..., `data[size-1]` and the remaining array elements `data[size]`, ..., `data[data.length-1]` are free for storing more elements. Therefore we choose the following instance data fields for the bag data.

```
private E[] data;
private int size;
```

As elements are added to the bag they are stored in the next available place in the array. Thus at any stage the array consists of two parts: the used part `data[0]` to `data[size-1]` and the unused part `data[size]` to `data[data.length-1]`.

Implementing the constructors

The first constructor implementation is

```
public FixedBag(int bagSize)
{
   data = (E[]) new Object[bagSize];
   size = 0;
}
```

and the second constructor calls this one. When constructing an array of generic type it is necessary to use the actual `Object` type for the array elements and typecast it to the type `E`. For various technical reasons related to the way generic types were added to the Java language the statement

```
data = (E[]) new E[bagSize];
```

is illegal.

Finally, the copy constructor is given by

```
public FixedBag(FixedBag<E> b)
{
   size = b.size();
   data = (E[]) new Object[b.data.length];
   for (int k = 0; k < size; k++)
      data[k] = b.data[k];
}
```

Here we first construct an array of the same maximum size `b.data.length` of the array `b`. Then the bag elements in `b` are copied into this array.

Implementing the methods

The `add` method first checks if there is room for the new element. Since `size` represents the number of elements currently in the `data` array then the new element can use `data[size]`. The implementation is

```
public boolean add(E element)
{
   if (size == data.length) // full bag
      return false;
   data[size] = element;
   size = size + 1;
   return true;
}
```

The `remove` method needs to use a loop to search for the element to remove. If the element is found at position `k` in the array then the obvious way to remove it is to use a for-loop to copy the array elements `data[k+1]`, ..., `data[size-1]` down one location to overwrite the element at position `k`. This requires another loop.

A more efficient way is to realize that a bag is not an ordered structure so the array ordering does not need to be preserved. Therefore we can just overwrite the element at position `k` with the last array element at position `size-1`. This effectively removes the element at position `k`. This gives the implementation

```
public boolean remove(E element)
{
   for (int k = 0; k < size; k++)
   {
      if (data[k].equals(element))
      {
         data[k] = data[size-1];
         size = size - 1;
         return true;
      }
   }
   return false; // not found
}
```

It is necessary to use the `equals` method defined for element type `E` to properly test for element equality. The test `data[k] == element` will not work. A class that does not have a properly defined `equals` method can not be used as the element type. The wrapper classes and the `String` class all have `equals` methods.

The remaining methods are easily implemented and the complete implementation class is

Class `FixedBag<E>`

`book-project/chapter13/bags`

```
package chapter13.bags;
/**
 * A simple fixed size bag implementation.
 * @param <E> type of elements in the bag
 */
public class FixedBag<E> implements Bag<E>
{
   // This version uses a fixed array for the bag

   private E[] data;
   private int size;

   /**
    * Create a bag for a given maximm number of elements.
    * @param bagSize the maximum number of elements
    */
   public FixedBag(int bagSize)
   {
      data = (E[]) new Object[bagSize];
      size = 0;
   }
```

```
/**
 * Create a default bag for a maximum of 10 elements
 */
public FixedBag()
{
   this(10);
}

/**
 * Construct a bag that is a copy of a given bag.
 * The copy has the same maximum size as bag b.
 * @param b the bag to copy
 */
public FixedBag(FixedBag<E> b)
{
   size = b.size();
   data = (E[]) new Object[b.data.length];
   for (int k = 0; k < size; k++)
      data[k] = b.data[k];
}

public int size()
{
   return size;
}

public boolean isEmpty()
{
   return size == 0;
}

public boolean add(E element)
{
   if (size == data.length)
      return false;
   data[size] = element;
   size = size + 1;
   return true;
}

public boolean remove(E element)
{
   for (int k = 0; k < size; k++)
   {
      if (data[k].equals(element))
      {
         // nice trick
         data[k] = data[size-1];
         size = size - 1;
         return true;
      }
      }
      return false; // not found
   }

   public boolean contains(E element)
   {
      for (int k = 0; k < size; k++)
         if (data[k].equals(element))
            return true;
      return false; // not found
   }

   /**
    * Return a string representation of this bag.
    * @return a string representation of this bag.
    */
   public String toString()
   {
      StringBuilder sb = new StringBuilder();
      sb.append("[");
      if (size != 0)
      {
         sb.append(data[0]);
         for (int k = 1; k < size; k++)
         {
            sb.append(",");
            sb.append(data[k]);
         }
      }
      sb.append("]");
      return sb.toString();
   }
}
```

We have not included comments for the interface methods since they are already given in the `Bag<E>` interface.

Converting to a dynamic implementation

We now convert the fixed size implementation to a dynamic one. This can easily be done by modifying the `add` method to automatically expand the `data` array whenever it it is full. The new version of `add` is

```
public boolean add(E element)
{
   if (size == data.length)
      resize();
   data[size] = element;
   size = size + 1;
   return true;
}
```

Here we call a `resize` method that increases the capacity as follows: (1) make a new data array twice the size of the current one, (2) copy the current data array to the beginning of the new one, (3) reassign the `data` reference to the new array (the old one will be garbage collected).

This gives the following private method.

```
private void resize()
{
   int newCapacity = 2 * data.length;
   E[] newData = (E[]) new Object[newCapacity]; // step 1
   for (int k = 0; k < data.length; k++) // step 2
      newData[k] = data[k];
   data = newData; // step 3
}
```

Here is the complete implementation.

Class `DynamicBag<E>`

`book-project/chapter13/bags`

```
package chapter13.bags;
/**
 * A simple dynamic bag implementation.
 * @param <E> the type of elements in the bag
 */
public class DynamicBag<E> implements Bag<E>
{
   private E[] data;
   private int size;

   /**
    * Create a bag with a given initial capacity.
    * @param initialCapacity the initial capacity of this bag
    */
   public DynamicBag(int initialCapacity)
   {
      data = (E[]) new Object[initialCapacity];
      size = 0;
   }

   /**
    * Create a default bag with an initial capacity of 10 elements.
    */
   public DynamicBag()
   {
      this(10);
   }

   /**
    * Construct a bag that is a copy of a given bag.
    * The copy has the same current maximum size as bag b.
    * @param b the bag to copy
    */
   public DynamicBag(DynamicBag<E> b)
   {
      size = b.size();
      data = (E[]) new Object[b.data.length];
      for (int k = 0; k < size; k++)
         data[k] = b.data[k];
   }

   public int size() {...} // same as for FixedBag
   public boolean isEmpty() {...} // same as for FixedBag

   public boolean add(E element)
   {
      if (size == data.length)
         resize();
      data[size] = element;
      size = size + 1;
      return true;
   }

   public boolean remove(E element) {...} // same as for FixedBag
   public boolean contains(E element) {...} // same as for FixedBag

   private void resize()
   {
      // Make a new array twice as big as current one,
      // copy data to it and make data reference the new one.

      int newCapacity = 2 * data.length;
      E[] newData = (E[]) new Object[newCapacity];
      for (int k = 0; k < data.length; k++)
         newData[k] = data[k];
      data = newData;
   }

   public String toString() {...} // same as for FixedBag
}
```

13.3.2 Implementation of the `DynamicArray` ADT

We have written a `FixedBag<E>` ADT but we will not consider a `FixedArray<E>` ADT since the built-in `array` type is a fixed size implementation.

Unlike a bag, an array is an ordered ADT. There is a first element, a second element, and so on so there is an index associated with each array element.

Designing the `Array` ADT

As for the Bag ADT we can use the following interface to design a simple array ADT

Interface `Array<E>`

`book-project/chapter13/arrays`

```
package chapter13.arrays;
/**
 * A simple generic array ADT.
 * @param <E> type of elements in the array
 */
public interface Array<E>
{
   /**
    * Return current number of elements in this array.
    * @return current number of elements in this array
    */
   int size();

   /**
    * Return true if this array is empty else false.
    * @return true if this array is empty else false
    */
   boolean isEmpty();

   /**
    * Add another element to end of this array.
    * @param element the element to add to end at position size().
    * @return true if add was successful else false.
    */
   boolean add(E element);

   /**
    * Get the element at a given index (0,1,...).
    * @param index the index of the element
    * @return the element at the index
    * @throws ArrayIndexOutOfBoundsException if the
    * index is out of the range 0 <= index < size()
    */
   E get(int index);

   /**
    * Set a new value for a given array element.
    * @param index the index of the array element
    * @param element the new value of the element
    * @throws ArrayIndexOutOfBoundsException if the
    * index is out of the range 0 <= index < size()
    */
   void set(int index, E element);
}
```

Here we have an `add` method that adds an element at the end of the array (position `size()`). It is important that we specify that the element be added at the end of the array. This was not necessary for the bag ADT.

The element at position `k` can be obtained using the `get` method and the `set` method can be used to give a new value to the object associated with position `k`. If an index `k` is outside the range `0 <= k < size()` then an `ArrayIndexOutOfBounds` exception is thrown.

The operations defined for an array ADT are quite different than those for a bag ADT since the array ADT is an ordered collection of elements and there is no assumed order for the elements in the bag. The `get` and `set` methods were not part of the bag ADT since there is no concept of an index for the elements in a bag.

This is a minimal array interface and there are many other methods such as a `remove` method that removes the element at a given index, and `indexOf` that returns the index of a given elememt.

Designing a dynamic implementation

The dynamic array implementation has the form

```
public class DynamicArray<E> implements Array<E>
{
   private E[] data;
   private int size;

   public DynamicArray(int initialCapacity) {...}
   public DynamicArray() {...}
   public DynamicArray(DynamicArray<E> a) {...}

   public int size() {...}
   public boolean isEmpty() {...}
   public boolean add(E element) {...}
   public E get(int index) {...}
   public void set(int index, E element) {...}
   public String toString() {...}
}
```

Here we use the same data structure, a fixed array, as for the bag implementations. The constructors are very similar to the `DynamicBag` constructors.

Using the design

Now we can try out some statements for our dynamic array design.

■ EXAMPLE 13.3 **(Resizing a dynamic array)** The following statements test that an array is resized when it becomes full. Autoboxing is used to convert integers to the `Integer` object type.

```
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1); a.add(2); a.add(3); a.add(4);
System.out.println("Array size is " + a.size());
System.out.println(a);
```

Here the initial capacity is 3. When we add the 4-th number the capacity is doubled to 6 and the number 4 is added to the array, which now has size 4 and room for two more elements. ■

■ EXAMPLE 13.4 **(Summing the elements in a dynamic array)** Unlike the bag we can loop over the elements in the array by using the `get` method. Here we construct an integer array and sum its elements using the following statements.

```
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1); a.add(2); a.add(3); a.add(4);
int sum = 0;
for (int k = 0; k < a.size(); k++)
   sum = sum + a.get(k);
System.out.println("The sum of the elements is " + sum);
```

Compare these statements with the following ones that do the same thing with a standard array:

```
int[] a = new int[4];
a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4;
int sum = 0;
for (int k = 0; k < a.length; k++)
   sum = sum + a[k];
```

Here we need to use the exact size of 4. ■

■ EXAMPLE 13.5 **(Swapping two elements of an array)** Assuming that `str` is an array of strings, the statements

```
String temp = str.get(i);
str.set(i, str.get(j));
str.set(j, temp);
```

swap the strings at positions `i` and `j`. ■

Implementing the constructors and methods

This is the same as for `DynamicBag<E>` and the implementation of the `get` and `set` methods are simple so we have the following class.

Class `DynamicArray<E>`

`book-project/chapter13/arrays`

```
package chapter13.arrays;
/**
 * A simple dynamic array implementation.
 * @param <E> type of elements in the array
 */
public class DynamicArray<E> implements Array<E>
{
   private E[] data;
   private int size;
```

```
/**
 * Create an array for a given initial capacity.
 * @param initialCapacity the initial capacity
 */
public DynamicArray(int initialCapacity)
{
   data = (E[]) new Object[initialCapacity];
   size = 0;
}

/**
 * Create a default array for an initial capacity of 10 elements.
 */
public DynamicArray()
{
   this(10);
}

/**
 * Construct an array that is a copy of a given array.
 * The copy has the same capacity as array a.
 * @param a the array to copy
 */
public DynamicArray(DynamicArray<E> a)
{
   size = a.size();
   data = (E[]) new Object[a.data.length];
   for (int k = 0; k < size; k++)
      data[k] = a.data[k];
}

public int size()
{
   return size;
}

public boolean isEmpty()
{
   return size == 0;
}

public boolean add(E element) {...} // same as for DynamicBag

public E get(int index)
{
   if (0 <= index && index < size)
      return data[index];
   else
      throw new ArrayIndexOutOfBoundsException("index out of bounds");
}

public void set(int index, E element)
```

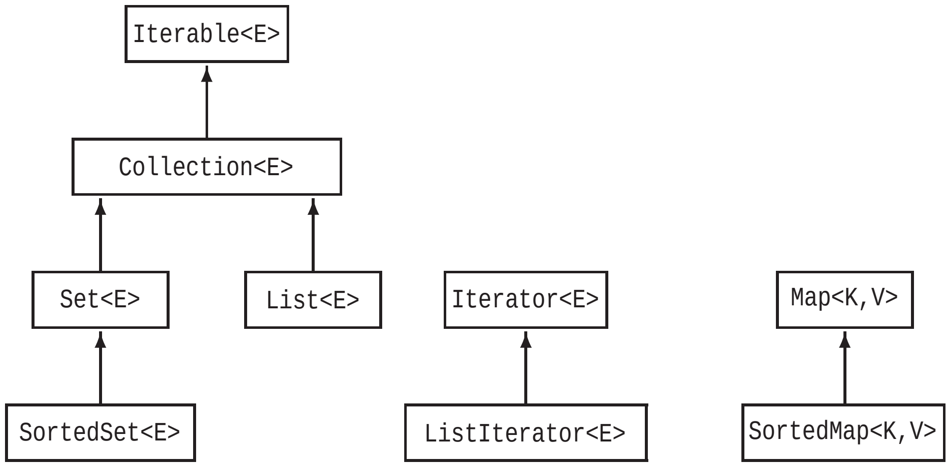


Figure 13.1: JCF related interface hierarchy

```
   {
      if (0 <= index && index < size)
         data[index] = element;
      else
         throw new ArrayIndexOutOfBoundsException("index out of bounds");
   }

   private void resize() {...} // same as for DynamicBag
   public String toString() {...} // same as for FixedBag
}
```

13.4 Java Collections Framework (JCF)

Many ADTs collect together elements of some data type. The simplest examples we have considered are the bag ADT and the array ADT. We define a **collection** as a data type that organizes a group of related objects called the elements of the collection and provides operations on them. There are often restrictions on the elements that belong to a specific kind of collection and on the way the elements can be accessed.

13.4.1 Interface hierarchy

In Java collections are represented by classes that implement the `Collection<E>` interface or one of its extended interfaces such as `Set<E>` or `List<E>`. These interfaces and others make up what is called the JCF (Java Collections Framework) and their relationship is shown in Figure 13.1. Here the arrow means “extends”. For example the `Set<E>` interface extends `Collection<E>`.

A set is an example of a collection whose elements have the following two properties: (1) no defined order and (2) duplicate elements are not allowed. This corresponds to the mathematical definition of a set.

```
public interface Collection<E> extends Iterable<E>
{
   // Query operations
   int size();
   boolean isEmpty();
   boolean contains(Object obj);
   Iterator<E> iterator();
   Object[] toArray();
   <T> T[] toArray(T[] a);

   // Modification Operations
   boolean add(E element); // optional
   boolean remove(Object obj); // optional

   // Bulk Operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); // optional
   boolean removeAll(Collection<?> c); // optional
   boolean retainAll(Collection<?> c); // optional
   void clear(); // optional

   // Comparison and hashing
   boolean equals(Object obj);
   int hashCode();
}
```

Figure 13.2: The `Collection<E>` interface

A bag is another example of a collection that, like a set, imposes no defined order on its elements but does allow duplicate elements. In mathematics a bag is called a multi-set. The bag is the simplest kind of collection class since it imposes no restrictions or structure on its elements.

Arrays and lists are collections in which the elements do have a defined order. There is a first element, a second element, and so on, and duplicates are allowed. In mathematics an array or list is often called a sequence.

We shall give a survey of the most important classes in the Java Collections Framework (JCF). Our goal is not to understand the implementation of these classes, which is left to a data structures course, but to learn how to use them. Of course, we should not need to understand implementation details in order to use a class.

The most important interface in the JCF is the `Collection<E>` interface which represents the basic design and methods any collection class should have. A class that implements this interface "is a" collection. A summary of this interface is given in Figure 13.2. It also extends another interface called `Iterable<E>`, given in Figure 13.3 and this interface contains one method called `iterator` which returns an object from a class that implements the `Iterator<E>` interface shown in Figure 13.4. We now discuss these three interfaces.

```
public interface Iterable<E>
{
   Iterator<E> iterator();
}
```

Figure 13.3: The `Iterable<E>` interface

```
public interface Iterator<E>
{
   boolean hasNext();
   E next();
   void remove(); // optional
}
```

Figure 13.4: The `Iterator<E>` interface

13.4.2 Traversing a collection with an iterator

An important operation on a collection is to be able to traverse it. This means to examine or process elements in the collection one at a time using some kind of loop. This is the purpose of an **iterator**.

Our simple `Bag<E>` interface did not define an iterator so for classes such as `FixedBag<E>` and `DynamicBag<E>` there was no way to process the elements one by one in some order. We could do this for the `DynamicArray<E>` class only because we had an indexed collection so we could use a standard for-loop to traverse an array as shown in Example 13.4.

In the JCF an iterator is an object of some class that implements the `Iterator<E>` interface shown in Figure 13.4. A collection class will normally not implement this interface directly. Instead it will provide an `iterator()` method that returns an object of some class that implements the `Iterator<E>` interface. This is the case for the `Collection<E>` interface shown in Figure 13.2 (under query operations).

In the `Iterator<E>` interface the `hasNext()` method is used to stop the iteration process and the `next()` method returns the current element in the collection and advances to the next one. This means that we can call `next()` repeatedly as long as `hasNext()` returns true.

■ EXAMPLE 13.6 **(Using an iterator to traverse a collection)** We can use statements such as the following to process the elements.

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   // do something here with element
}
```

Here `ACollectionClass` is any class that implements the `Collection<E>` interface. ■

The `Iterator<E>` interface also contains a `remove` operation which is listed as optional. If an implementing class does not support the removal of elements from the collection then an `UnsupportedOperationException` will be thrown. Such an iterator is said to be immutable.

■ EXAMPLE 13.7 **(Using an iterator as a filter)** The following statements show how an iterator can be used as a **filter** by removing elements from the collection that satisfy some condition.

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   if (removal condition is true)
   {
      iter.remove();
   }
}
```

Here it is important that the `remove()` method is used after a call to `next()`. ■

■ EXAMPLE 13.8 **(Using an iterator as a filter without remove)** If removal is not supported then a filter can be written by creating a new collection containing only the elements that were not removed:

```
Collection<E> c = new ACollectionClass<E>(...);
c.add(e1); c.add(e2); c.add(e3); // ...
// create a new empty collection
Collection<E> newCollection = new ACollectionClass<E>();
Iterator<E> iter = c.iterator();
while(iter.hasNext())
{
   E element = iter.next();
   if (removal condition is NOT true)
      newCollection.add(element);
}
```

Here the original collection is not changed. ■

An important property of an iterator is that it does not expose any internal details of the collection and the data structures used in the implementation. This is important since it means that the implementation of the collection class can be changed without changing the iterator.

13.4.3 `Iterable<E>` interface

The `Iterable<E>` interface is related to the for-each loop introduced in Java 5. If a class implements this interface then it provides an `iterator()` method defining an iterator and the for-each loop can be applied as follows

■ EXAMPLE 13.9 **(Using a for-each loop as an immutable iterator)** The for-each loop has the syntax

```
for (E element : c)
{
   // do something here with element
}
```

Here `c` is any object from a class that implements the `Iterable<E>` interface. In particular it can be of type `Collection<E>`. The for-each loop cannot access the `remove()` method so it can only be used for immutable traversals. ■

■ EXAMPLE 13.10 **(Using an iterator with a standard array type)** The built-in array type also implements `Iterable<E>` so it is possible to process an array using statements such as

```
String[] s = new String[3];
s[0] = "one"; s[1] = "two"; s[2] = "three";
for (String str : s)
{
   // do something here with the string str
}
```

This is useful as a replacement for the standard for-loop that does not actually use its index in the body of the loop. The for-each loop requires no index. ■

13.5 `Collection<E>` and `Set<E>` interfaces

13.5.1 `Collection<E>` interface

We now summarize the methods in the `Collection<E>` interface in Figure 13.2. For more complete descriptions see the Java API documentation. As shown in the figure the operations can be divided into four categories: (1) Query operations, (2) Modification operations, (3) Bulk operations, and (4) Comparison and hashing.

Some methods are optional. If a class does not want to implement an optional method the method must throw an `UnsupportedOperationException` if it is called. Note that the optional operations are precisely the ones which may modify this collection, so if a class implements none of these methods then it is implementing immutable collections. Here is a summary of the `Collection<E>` methods.

Note that the `contains` and `remove` methods have an argument of type `Object` instead of `E`. This is conventional since these methods do not add new elements to the collection. However, the `add` method must have an argument of type `E` to guarantee that the collection will only contain elements of type `E`.

- **`int size()`**

  Return the number of elements in **`this`** collection.

- **`boolean isEmpty()`**

  Returns true if there are no elements in **`this`** collection else returns false.

- **`boolean contains(Object obj)`**

  Returns true if **`this`** collection contains element **`obj`** else returns false.

- **`Iterator<E> iterator()`**

  Return an iterator of type **`Iterator<E>`** for **`this`** collection. This is the method that is necessary to implement the **`Iterable<E>`** interface.

- **`Object[] toArray()`**

  Convert the elements in **`this`** collection to an array of **`Object`** type.

  For example, if `c` is a collection of strings then the statement

  ```
  Object[] s = c.toArray();
  ```

  converts the collection of strings to the array `s` of objects `s[0]`, ..., `s[s.length-1]`. To recover the strings it is necessary to use a typecast on each component such as

  ```
  String str = (String) s[k];
  ```

- **`<T> T[] toArray(T[] a)`**

  This is a parametrized method for type **`T`** that returns an array **`T[]`** of type **`T`**.

  If the parametrized type of the collection is **`T`** as indicated by the argument **`a`** then this method converts the elements of **`this`** collection to an array of type **`T`** which is the run-time type of the array. If the collection does not contain elements of type **`T`** an exception is thrown.

  For example, if `c` is a collection of strings then the statement

  ```
  String[] s = c.toArray(new String[c.size()]);
  ```

  converts the collection of strings to the array `s` of strings `s[0]`, ..., `s[s.length-1]`.

- **`boolean add(E element)`**

  Returns true if **`this`** collection was changed (**`element`** was added) after calling the method else returns false. This is an optional operation.

- **`boolean remove(Object obj`**

  Returns true if **`this`** collection was changed (**`obj`** was found and removed) after calling the method else returns false. This is an optional operation.

```
public interface Set<E> extends Collection<E>
{
   // The Collection<E> interface methods can go here
   // The Set<E> interface introduces no new methods
}
```

Figure 13.5: The `Set<E>` interface

- **`boolean containsAll(Collection<?> c)`**

  Returns true if **`this`** collection contains all the elements in collection **`c`** else returns false. The notation **`Collection<?>`** means a collection of any type (**`?`** is a wild card).

- **`boolean addAll(Collection<? extends E> c)`**

  Adds all of the elements of **`c`** to **`this`** collection. Returns true if **`this`** collection was modified after calling the method else returns false. The notation **`Collection<? extends E>`** means a collection of any type that extends or implements the type **`E`**. In this context **`extends`** means "extends or implements". This is an optional operation.

- **`boolean removeAll(Collection<?> c)`**

  Returns true if **`this`** collection was modified (one or more elements of **`c`** were removed from **`this`** collection) after calling the method else returns false. This is an optional operation.

- **`boolean retainAll(Collection<?> c)`**

  Retains only the elements in **`this`** collection that are also in **`c`**. Returns true if this collection was modified after calling the method else returns false. This is an optional operation.

- **`void clear()`**

  Remove all elements of **`this`** collection to give an empty collection. This is an optional operation.

- **`boolean equals(Object obj)`**
  `int hashCode()`

  These are methods in the **`Object`** class that can be overridden. The **`equals`** method tests if two collections have the same elements.

13.5.2 `Set<E>` interface

The `Collection<E>` interface describes what is called a bag or multi-set since there is no structure imposed on the elements in the collection.

The `Set<E>` interface is given in Figure 13.5. It extends `Collection<E>` but does not introduce any new methods. However the documentation of some of the methods changes since a set is a collection that does not contain duplicates. For example, the `contains` method will return false if the element `obj` is already in `this` set and the `add` method will not change the collection if the element `obj` is already in `this` set.

Similarly the `addAll` method will only add to `this` set the elements of the collection `c` that are not already in `this` set.

Set theory interpretation of the bulk set methods

The bulk `Set<E>` methods can be used to implement the basic set theory operations of subset, set difference, intersection, and union.

subset/superset If a and b are two sets then $a \subseteq b$ (or equivalently $b \supseteq a$) means that a is a subset of b (or equivalently b is a superset of a). In other words every element in a is also an element of b.

This can be expressed using `containsAll`. If `a` and `b` are two sets (objects from a class that implements `Set<E>`) then `a.containsAll(b)` returns true only if $a \supseteq b$, so `containsAll` is the superset operation.

set difference If a and b are two sets then $a - b$ is the difference: set of all elements in a that are not in b. A destructive version is represented by `a.removeAll(b)`, which replaces a by $a - b$.

set union If a and b are two sets then $a \cup b$ is their union: set of all elements in a or b or both. A destructive version is represented by `a.addAll(b)`, which replaces a by $a \cup b$.

set intersection If a and b are two sets then $a \cap b$ is their intersection: set of all elements that are in a and in b. A destructive version is represented by `a.retainAll(b)`, which replaces a by $a \cap b$.

To obtain non-destructive versions (`a` is not changed) it is necessary to make a copy of `a` and apply the operation to the copy.

13.6 Set Implementations and examples

The JCF includes several implementations of the `Set<E>` interface. We will consider three of them: `HashSet<E>`, `LinkedHashSet<E>`, and `TreeSet<E>`. The `HashSet<E>` implementation is the fastest but if a total order can be defined on the elements of the set then `TreeSet<E>` can be used to maintain the set in sorted order unlike `HashSet<E>` which maintains no order. If the element order is not important use `HashSet<E>`. The `LinkedHashSet<E>` class maintains the elements in the order they were added to the set.

13.6.1 `HashSet<E>` implementation of `Set<E>`

A summary of the `HashSet<E>` implementation is given in Figure 13.6. We will not discuss any implementation details. There are four constructors. The first constructor with no arguments

```
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>, Cloneable, Serializable
{
   public HashSet() {...}
   public HashSet(int initialCapacity) {...}
   public HashSet(Collection<? extends E> c) {...}
   public HashSet(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // implementations of Set interface methods go here
}
```

Figure 13.6: The `HashSet<E>` class

```
public class LinkedHashSet<E> extends HashSet<E>
   implements Set<E>, Cloneable, Serializable
{
   public LinkedHashSet() {...}
   public LinkedHashSet(int initialCapacity) {...}
   public LinkedHashSet(Collection<? extends E> c) {...}
   public LinkedHashSet(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // implementations of Set interface methods go here
}
```

Figure 13.7: The `LinkedHashSet<E>` class

constructs an empty set with a default initial capacity of 16 elements. The second constructor specifies a given initial capacity.

The third one is called a **conversion constructor** and is very useful. It creates a set of element type `E` from any given collection `c` which may have any element type which extends or implements the type `E`. This constructor can also be used as a copy constructor if `c` has type `E`.

We will not use the fourth constructor. It is used to optimize the hash table implementation.

13.6.2 `LinkedHashSet<E>` implementation of `Set<E>`

A summary of the `LinkedHashSet<E>` implementation is given in Figure 13.7. The constructors are identical to the ones in `HashSet<E>`.

13.6.3 `TreeSet<E>` implementation of `SortedSet<E>` and `Set<E>`

A summary of the `TreeSet<E>` implementation is given in Figure 13.8. Note that `TreeSet<E>`

```
public class TreeSet<E> extends AbstractSet<E>
   implements SortedSet<E>, Cloneable, Serializable
{
   public TreeSet() {...}
   public TreeSet(Collection<? extends E> c) {...}
   public TreeSet(Comparator<? super E> c) {...}
   public TreeSet(SortedSet<E> s){...}

   public Object clone() {...}

   // implementations of SortedSet interface methods go here
   // SortedSet extends the Set interface
}
```

Figure 13.8: The `TreeSet<E>` class

implements the `SortedSet<E>` interface which extends the `Set<E>` interface so `TreeSet<E>` also extends `Set<E>`. We will not need the extra methods provided by the `SortedSet<E>` interface.

There are four constructors. The first provides an empty set. As elements are added they will sorted according to the natural order of the elements of type `E` (`E` must implement `Comparable<E>`).

The second is a **conversion constructor** similar to the one in `HashSet<E>`. It creates a sorted set of element type `E` from any given collection `c` which may have type `E` or any element type which extends or implements the type `E`.

The third constructor provides a `Comparator` argument which has type `E` or any type that is a super type of `E`. It's purpose is to define the total order to be used by `TreeSet<E>`. If this constructor is not used then the natural ordering defined by the element type `E` is used. In this case the type `E` must implement the `Comparable<E>` interface.

The last constructor is a copy constructor which makes a copy of any sorted set.

13.6.4 Simple set examples

■ EXAMPLE 13.11 **(Removing duplicates from a collection)** Suppose we have a collection `c` of strings and we want to obtain a new collection that is `c` with duplicates removed. The following statement does this

```
Set<String> noDups = new HashSet<String>(c);
```

using the conversion constructor. ■

■ EXAMPLE 13.12 **(Random sets of elements)** The following statements create a set of 10 integers generated randomly in the range 1 to n where $n > 9$.

```
Random random = new Random();
Set<Integer> randomSet = new TreeSet<Integer>();
while (randomSet.size() < 10)
{
   randomSet.add(random.nextInt(n) + 1);
}
```

Here we simply try to add elements until the set has size 10. It is important to have $n > 9$ or the loop will be infinite since there are no sets of size 10 containing only numbers in the range $1 \le k \le 9$. ■

■ EXAMPLE 13.13 **(Using HashSet to compute set union)** The statements

```
Set<String> s1 = new HashSet<String>();
s1.add("one"); s1.add("two"); s1.add("three");
Set<String> s2 = new HashSet<String>();
s2.add("four"); s2.add("five"); s2.add("six");
```

define two sets of strings and the statements

```
Set<String> union = new HashSet<String>(s1);
union.addAll(s2);
System.out.println(union);
```

create a copy of `s1` and use `addAll` to compute the union of the two sets without modifying either `s1` or `s2`. The result displayed is

```
[one, two, five, four, three, six]
```

The output shows there is no specific order.

If you replace `HashSet` by `LinkedHashSet` everywhere the result displayed is

```
[one, two, three, four, five, six]
```

Now the order is the same as the order in which the strings were added to the set.

If you replace `HashSet` by `TreeSet` everywhere the result displayed is

```
[five, four, one, six, three, two]
```

Now the elements appear in alphabetical order. ■

■ EXAMPLE 13.14 **(Using an iterator as a filter)** The statements

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(3); s.add(4); // [1,2,3,4]
Iterator<Integer> iter = s.iterator(); // ask s for an iterator
while (iter.hasNext())
{
   int k = iter.next();
   if (k % 2 == 0)
      iter.remove();
}
System.out.println(s);
```

use an iterator to remove all the even integers from the set `s` of integers. The print statement displays `[1,3]`. ■

■ EXAMPLE 13.15 **(Use an iterator as a filter)** The following statements

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(3); s.add(4); // [1,2,3,4]
Iterator<Integer> iter = s.iterator(); // ask s for an iterator
Set<Integer> evenSet = new HashSet<Integer>();
Set<Integer> oddSet = new HashSet<Integer>();
while (iter.hasNext())
{
   int k = iter.next();
   if (k % 2 == 0)
      evenSet.add(k);
   else
      oddSet.add(k);
}
System.out.println(evenSet);
System.out.println(oddSet);
```

use an iterator to create two new sets from `s`, one containing the even integers in `s` and the other containing the odd integers in `s`. The print statements display `[2,4]` and `[1,3]` ■

13.6.5 Removing duplicates from a list of words

Using sets we can easily write a program that removes duplicate words in a list of words. Simply read the words and add them to a set. Any duplicates will not be added.

Class `RemoveDuplicateWords`

`book-project/chapter13/sets`

```
package chapter13.sets;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;
import java.util.Set;

/**
 * Remove duplicate words from a file of words.
 */
public class RemoveDuplicateWords
{
   public void doTest() throws FileNotFoundException
   {
      Scanner input = new Scanner(new File("files/words.txt"));
      Set<String> uniqueSet = new HashSet<String>();
      Iterator<String> iter = input;
      while(iter.hasNext())
      {
         uniqueSet.add(iter.next());
      }
      input.close();
      System.out.println(uniqueSet.size() + " unique words found:");
      System.out.println(uniqueSet);
   }

   public static void main(String[] args) throws FileNotFoundException
   {
      new RemoveDuplicateWords().doTest();
   }
}
```

Here we use the fact that the `Scanner` class implements the `Iterator<String>` interface. As each word is read an attempt is made to add it to the set. You can try this program using a file such as

```
all all
words words
are are duplicated duplicated
```

The output is

```
4 unique words found:
[words, all, duplicated, are]
```

You may get a different order since we are using a `HashSet`. For output in alphabetic order use `TreeSet`. For a related problem see Exercise 13.10.

13.7 `List<E>` and `ListIterator<E>` interfaces

A list is a collection of elements arranged in some linear order. It has a first element, a second element and so on. According to Figure 13.1 the `List<E>` interface extends `Collection<E>` so you can think of a list as an ordered collection of elements. The `List<E>` interface is summarized in Figure 13.9. As for the `Collection<E>` interface the operations that can modify a list are indicated as optional so an implementation for immutable lists would not implement these operations.

For traversing lists the `Iterator<E>` interface has been extended to provide a two way iterator called `ListIterator<E>` summarized in Figure 13.10.

13.7.1 `List<E>` interface

The methods from the `Collection<E>` class have basically the same meaning in the `List<E>` interface except that the `add` and `addAll` methods now specify that these operations append the elements to the end of the list and the `remove` method specifically removes the first occurrence of the element.

```
public interface List<E> extends Collection<E>
{
   // The Collection<E> interface methods can go here

   // Positional Access Operations
   E get(int index);
   E set(int index, E element); // optional
   void add(int index, E element); // optional
   E remove(int index); // optional
   boolean addAll(int index, Collection<? extends E> c); // optional

   // Search Operations
   int indexOf(Object obj);
   int lastIndexOf(Object obj);

   // List Iterators
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

   // View
   List<E> subList(int fromIndex, int toIndex);
}
```

Figure 13.9: The `List<E>` interface

```
public interface ListIterator<E> extends Iterator<E>
{
   // Query Operations
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();

   int nextIndex();
   int previousIndex();

   // Modification Operations
   void remove(); // optional
   void set(E element); // optional
   void add(E element); // optional
}
```

Figure 13.10: The `ListIterator<E>` interface

We now summarize the extra methods introduced by the `List<E>` interface of Figure 13.9. The additional methods fall into four categories: (1) positional access operations that locate list elements using an index, (2) search operations that find a list element given its index, (3) list iterators that begin at the start of a list or at some other position, and (4) a view operation that returns a sublist.

- **`E get(int index)`**

  Return the element in **`this`** list at position given by **`index`**. If **`index < 0`** or **`index >= size()`** an index out of bounds exception is thrown.

- **`E set(int index, E element)`**

  Replace the element at position **`index`** by the given element. The element being replaced is returned. If **`index < 0`** or **`index >= size()`** an `IndexOutOfBoundsException` is thrown. This is an optional operation.

- **`void add(int index, E element)`**

  Add a new element to **`this`** list at position **`index`**. The elements originally beginning at position **`index`** are moved up to higher indices to accommodate the new element. If **`index < 0`** or **`index > size()`** an index out of bounds exception is thrown. Note that **`index = size()`** is allowed here, corresponding to adding after the last element. This is an optional operation.

- **`boolean addAll(int index, Collection<? extends E> c)`**

  Add all the elements in the given collection **`c`** to **`this`** list beginning at the given position **`index`**. The elements originally beginning at position **`index`** are moved up to higher indices to accommodate the new elements. The restrictions on **`index`** are the same as for the **`add`** method. This is an optional operation.

- **`int indexOf(Object obj)`**

  Return the index of the first occurrence of the given object **`obj`** in **`this`** list. If **`obj`** was not found then -1 is returned.

- **`int lastIndexOf(Object obj)`**

  Return the index of the last occurrence of the given object **`obj`** in **`this`** list. If **`obj`** was not found then -1 is returned.

- **`ListIterator<E> listIterator()`**
  `ListIterator<E> listIterator(int index)`

  Returns a **`ListIterator<E>`** object. For the no-arg version the iterator will start at the beginning of **`this`** list. The second version will start at position **`index`** in **`this`** list. The restrictions on **`index`** are the same as for **`get`**.

- **`List<E> subList(int fromIndex, int toIndex)`**

  Returns a sublist of `this` list beginning and ending at the given indices. If the indices are not in range an `IndexOutOfBoundsException` is thrown.

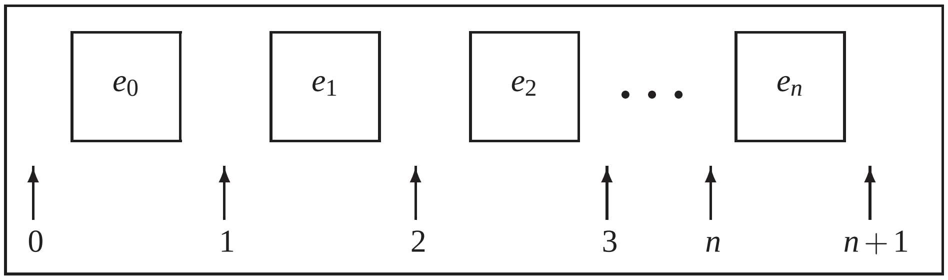


Figure 13.11: Indices for the list $[e_0, e_1, e_2, \ldots, e_n]$ lie between elements.

13.7.2 `ListIterator<E>` interface

As shown in Figure 13.10 the `ListIterator<E>` interface extends `Iterator<E>` so that the list can be traversed in either direction. The `Iterator<E>` part provides iteration in the forward direction using `hasNext()` and `next()` and the new methods provide iteration in the backward direction using `hasPrevious()` and `previous()`.

During iteration the `add`, `remove`, and `set` methods are available. They operate on the current element of the list (last element returned by `next()` or `previous()`). For `add` the element is inserted immediately before the next element that would be returned by `next()`, if any, and after the next element that would be returned by `previous()`.

When using a list iterator it is helpful to think of list indices as lying between the list elements as shown in Figure 13.11. Thus, a call to `next()` returns the element to the right of the index and advances to the next higher index. Similarly, a call to `previous()` returns the element to the left of the index and advances to the next lower index.

13.8 `List<E>` implementations and examples

The JCF includes two general purpose implementations of the `List<E>` interface: `ArrayList<E>` and `LinkedList<E>`.

13.8.1 `ArrayList<E>` implementation of `List<E>`

The `ArrayList<E>` class implements a dynamic array ADT and is the best implementation if you need positional access to the list using a 0-based index. Accessing an element given its index is an $O(1)$ operation. Thus this is a random access structure like the built-in `array` class. The `ArrayList<E>` class is summarized in Figure 13.12.

There are three constructors. The no-arg constructor provides a resizable list with initial space for 10 elements and the second constructor provides a resizable list with the specified initial capacity.

The third constructor is a conversion constructor that creates an `ArrayList<E>` from the given collection `c` in the order defined by the collection's iterator.

The dynamic increase in the size of the list occurs automatically as needed. Two methods are

```
public class ArrayList<E> extends AbstractList<E>
   implements List<E>, RandomAccess, Cloneable, Serializable
{
   // Constructors
   public ArrayList() {...}
   public ArrayList(int initialCapacity) {...}
   public ArrayList(Collection<? extends E> c) {...}

   // Implementation of List<E> interface methods go here

   // Extra methods
   public Object clone() {...}
   public void ensureCapacity(int minCapacity) {...}
   public void trimToSize() {...}
}
```

Figure 13.12: The `ArrayList<E>` class

supplied for resizing the list under program control. The `ensureCapacity` method can be used to expand the size to a specified amount if necessary and the `trimToSize` method can be used to downsize the list so that its capacity is the same as its size.

Our `DynamicArray<E>` class (see page 740) is a very simple version of `ArrayList<E>`.

13.8.2 `LinkedList<E>` implementation of `List<E>`

The `LinkedList<E>` implementation uses a linked list data structure (discussed in a data structures course). For random access (using an index) this implementation is inefficient ($O(n)$). If you mostly want to add and remove elements using the list iterator (access relative to the current element) then this implementation is efficient ($O(1)$) whereas the `ArrayList<E>` implementation would be inefficient. The `LinkedList<E>` class is summarized in Figure 13.13.

There are two constructors. The no-arg constructor creates an empty list. There is no need to specify a capacity since one of the properties of a linked list is that it can grow and shrink one element at a time in a very efficient manner.

The second constructor is a conversion constructor that creates a linked list from the given collection `c` in the order defined by the collection's iterator.

13.8.3 Simple list examples

■ EXAMPLE 13.16 **(Converting a collection to a list)** The statement

```
List<String> list = new ArrayList<String>(c);
```

uses the conversion constructor to convert any collection `c` of strings to an `ArrayList` of strings in the order given by the collection's iterator. ■

■ EXAMPLE 13.17 **(Appending to a list)** The statement

```
public class LinkedList<E> extends AbstractSequentialList<E>
   implements List<E>, Queue<E>, Cloneable, Serializable
{
   // Constructors
   public LinkedList() {...}
   public LinkedList(Collection<? extends E> c) {...}

   // Implementation of List<E> interface methods go here
   // Queue<E> related methods go here

   // Extra methods
   public Object clone() {...}
   public void addFirst(E element) {...}
   public void addLast(E element) {...}
   public E getFirst() {...}
   public E getLast() {...}
   public E removeFirst() {...}
   public E removeLast() {...}
}
```

Figure 13.13: The `LinkedList<E>` class

```
list1.addAll(list2);
```

appends `list2` to the end of `list1`.

The statements

```
List<String> list3 = new ArrayList<String>(list1);
list3.addAll(list2);
```

append two lists to create a new list without modifying either `list1` or `list2`. ■

■ EXAMPLE 13.18 **(Swapping (exchanging) two list elements)** Given a list of strings the following statements

```
String temp = list.get(i);  // String temp = list[i];
list.set(i, list.get(j));   // list[i] = list[j];
list.set(j, temp);          // list[j] = temp;
```

use the indexed list operations `get` and `set` to swap the elements at positions `i` and `j`. The comments show the statements that would be used if `list` were an array instead of a list.

The polymorphic static method

```
public static <E> void swap(List<E> list, int i, int j)
{
   E temp = list.get(i);
   list.set(i, list.get(j));
   list.set(j, temp);
   }
```

can be used to swap two elements of any list. ■

13.8.4 Book inventory example

Here we create a simple book inventory system. Each book is represented as an object from a `Book` class and the books in the store are represented as a list of type `ArrayList<Book>`,

Each book has data fields for a title, author, price, and the number of books in stock. We want to process a list of books and remove books that are not in stock. The books removed can be stored in another reorder list. The `Book` class is given by

Class Book

book-project/chapter13/lists

```
package chapter13.lists;
/**
 * Book objects have a title, author, price, quantity in stock.
 * Books can also be ordered by increasing order of title.
 */
public final class Book implements Comparable<Book>
{
   private String title;
   private String author;
   private double price;
   private int inStock;

   /**
    * Construct a book from given data.
    * @param title the title of the book.
    * @param author the author of the book.
    * @param price the retail price of the book.
    * @param inStock the number of books in stock.
    */
   public Book(String title, String author, double price, int inStock)
   {
      this.title = title;
      this.author = author;
      this.price = price;
      this.inStock = inStock;
   }

   /**
    * Return the author of the book.
    * @return the author of the book.
    */
   public String getAuthor()
   {
      return author;
   }
```

```
/**
 * Return the number of books in stock.
 * @return the number of books in stock.
 */
public int getInStock()
{
   return inStock;
}

/**
 * Return the retail price of the book.
 * @return the retail price of the book.
 */
public double getPrice()
{
   return price;
}

/**
 * Return the title of the book.
 * @return the title of the book.
 */
public String getTitle()
{
   return title;
}

/**
 * Return a string representation of a book.
 * @return a string representation of a book.
 */
public String toString()
{
   return "Book[" + title + "," +
      author + "," + price + "," + inStock + "]";
}

/**
 * Compare this book to another book using the title.
 * @param b the book to compare with this book
 * @return negative, zero, positive results
 */
public int compareTo(Book b)
{
   return title.compareTo(b.title);
}

/**
 * Return true if this book has the same title as obj.
 * @param obj the book to compare with this book
 * @return true if this book has same title as obj
```

```
    */
   public boolean equals(Object obj)
   {
      if (obj == null || getClass() != obj.getClass())
         return false;
      return title.equals(((Book) obj).title);
   }

   public int hashCode()
   {
      return title.hashCode();
   }
}
```

We have implemented the `Comparable<Book>` interface that defines the natural order with the `compareTo` method to be alphabetical order by title. An `equals` method has also been provided and the corresponding `hashCode` is obtained using the hash code of the title string. Choosing hash codes is best left to a course on data structures. Here we use the hash code already defined in the `String` class.

The following static method can be used to produce the two lists.

```
public static List<Book> reOrderBooks(List<Book> list)
{
   List<Book> reOrderList = new LinkedList<Book>();
   Iterator<Book> iter = list.iterator();
   while (iter.hasNext())
   {
      Book b = iter.next();
      if (b.getInStock() == 0)
      {
         reOrderList.add(b);
         iter.remove();
      }
   }
   return reOrderList;
}
```

Here `list` is the given list to split. A `reOrderList` is created and the iterator `iter` is used to traverse the given list, removing elements with an in stock value of 0. Each element removed is added to `reOrderList` which is returned by the method. Note that we have used `Iterator<Book>` instead of `ListIterator<Book>` since the extra methods in `ListIterator<Book>` are not used here.

We have used `LinkedList` here instead of `ArrayList` since we access the list only relatively using the iterator's `add` and `remove` methods which are efficient.

Here is a short program that can be used to test the method.

Class `BookList`

`book-project/chapter13/lists`

```
package chapter13.lists;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

public class BookList
{
   /**
    * Modify original list so it contains only books
    * in stock and create a new list that contains books
    * which are out of stock.
    */
   public void processBookList()
   {
      List<Book> list = new LinkedList<Book>();
      list.add(new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      list.add(new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      list.add(new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      list.add(new Book("Original Sin", "P. D. James", 39.95 ,0));
      list.add(new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));

      List<Book> reOrderList = reOrderBooks(list);
      System.out.println("Re-order list:");
      displayList(reOrderList);
      System.out.println("List in stock:");
      displayList(list);
   }

   /**
    * Create lists of books in stock and reorder list.
    * @param list the book list
    * @return the list of books to be ordered.
    * The original list now contains only books that are instock.
    */
   public static List<Book> reOrderBooks(List<Book> list)
   {
      List<Book> reOrderList = new LinkedList<Book>();
      Iterator<Book> iter = list.iterator();
      while (iter.hasNext())
      {
         Book b = iter.next();
         if (b.getInStock() == 0)
         {
            reOrderList.add(b);
            iter.remove();
         }
      }
      return reOrderList;
   }

   public static <E> void displayList(List<E> list)
   {
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      BookList books = new BookList();
      books.processBookList();
   }
}
```

A for-each loop is used to display the books, one per line and the output is

```
Re-order list:
Book[Stranger House,Reginald Hill,29.5,0]
Book[Original Sin,P. D. James,39.95,0]
Book[Fleshmarket Close,Ian Rankin,25.0,0]
List in stock:
Book[Dead Souls,Ian Rankin,25.95,10]
Book[Not Safe After Dark,Peter Robinson,32.99,10]
```

13.8.5 Insertion in a sorted list

An easy way to maintain a list in some sorted order is to start with an empty list and as elements are added to the list put them in the correct position so that the list remains sorted. In this way we avoid sorting altogether.

To develop the algorithm suppose that $[e_0, e_1, \ldots, e_n]$ is a list that is sorted in some order. If we want to add an element e to the list in its proper sorted position then we need to iterate through the list and compare e with each e_k. The iteration continues until we arrive at an element e_k such that $e \le e_k$. Then the proper place for e is before e_k. There are two special cases: (1) list is empty so create a one-element list, (2) we never find that $e \le e_k$ so the element e must be added at the end of the list.

Let us assume that we have a sorted list of integers. Then we can write the following method to do the insertion.

```
public static void
insertInSortedIntegerList(List<Integer> list, Integer newElement)
{
   ListIterator<Integer> iter = list.listIterator();

   if (!iter.hasNext()) // empty list so make a 1-element list
   {
      iter.add(newElement);
      return;
   }

   while(iter.hasNext())
   {
      int ek = iter.next();
      if (newElement <= ek)
      {
         iter.previous(); // backup
         iter.add(newElement);
         return;
      }
   }
   iter.add(newElement); // add after end of list
}
```

It is important to note that `previous()` is needed since to find the correct position using `next()` we need to add the element at the position to its left so `previous()` backs up the iterator. If we come out of the while loop then we need to add the new element to the end of the list.

Statements such as the following can be used to test the method:

```
   List<Integer> list = new ArrayList<Integer>();
   list.add(4); list.add(6); list.add(8);
   System.out.println(list);
   insertInSortedIntegerList(list,9);
   System.out.println(list);
```

The result is the list `[4,6,8,9]`.

We can convert this method to the following polymorphic generic one with type `E`.

```
   public static <E extends Comparable<E>>
   void insertInSortedList(List<E> list, E newElement)
   {
      ListIterator<E> iter = list.listIterator();

      if (!iter.hasNext()) // empty list so make a 1-element list
      {
         iter.add(newElement);
         return;
      }

      while(iter.hasNext())
      {
         E element = iter.next();
         if (newElement.compareTo(element) <= 0)
         {
            iter.previous(); // backup
            iter.add(newElement);
            return;
         }
      }
      iter.add(newElement); // add after end of list
    }
```

Here we specify that the generic type must extend or implement the `Comparable<E>` interface. Then instead of using `<=` we use the `compareTo` method of the `Comparable<E>` interface.

This example can also be done using a `LinkedList<E>`, which may be more efficient than an `ArrayList<E>` in this case, since any modifications to the input list are done using only relative access and the list iterator operations are $O(1)$.

Here is a short program that can be used to test the method for lists of type `String` and `Book` both of which implement the `Comparable` interface.

Class `SortedListExample`

book-project/chapter13/lists

```
package chapter13.lists;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class SortedListExample
{
   public void doTest()
   {
      // Try it on a list of strings

      List<String> strList = new ArrayList<String>();
      strList.add("Fred"); strList.add("Jane"); strList.add("Mike");
      System.out.println(strList);
      insertInSortedList(strList, "Gord");
      System.out.println(strList);
      insertInSortedList(strList,"Carol");
      System.out.println(strList);
      insertInSortedList(strList,"Bob");
      System.out.println(strList);
      insertInSortedList(strList,"Susan");
      System.out.println(strList);

      // Try it on a list of books

      List<Book> list = new ArrayList<Book>();
      insertInSortedList(list, new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      insertInSortedList(list, new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      insertInSortedList(list,
                        new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      insertInSortedList(list, new Book("Original Sin", "P. D. James", 39.95 ,0));
      insertInSortedList(list, new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));
      displayList(list);
   }

   public static <E extends Comparable<E>>
   void insertInSortedList(List<E> list, E newElement)
   {
      ListIterator<E> iter = list.listIterator();
      if (!iter.hasNext()) // empty list so make a 1-element list
      {
         iter.add(newElement);
         return;
      }
      // Note: when we know where to insert
      // the new element we have gone one
      // position too far so previous is needed.
      while(iter.hasNext())
      {
         E element = iter.next();
         if (newElement.compareTo(element) <= 0)
         {
            iter.previous(); // backup
            iter.add(newElement);
            return;
         }
      }
      iter.add(newElement); // add after end of list
   }

   public static <E> void displayList(List<E> list)
   {
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      SortedListExample example = new SortedListExample();
      example.doTest();
   }
}
```

The sorted output is

```
      [Fred, Jane, Mike]
      [Fred, Gord, Jane, Mike]
      [Carol, Fred, Gord, Jane, Mike]
      [Bob, Carol, Fred, Gord, Jane, Mike]
      [Bob, Carol, Fred, Gord, Jane, Mike, Susan]
      Book[Dead Souls,Ian Rankin,25.95,10]
      Book[Fleshmarket Close,Ian Rankin,25.0,0]
      Book[Not Safe After Dark,Peter Robinson,32.99,10]
      Book[Original Sin,P. D. James,39.95,0]
      Book[Stranger House,Reginald Hill,29.5,0]
```

13.9 Map data type

Maps are one of the most important data types. A map is a function f that associates elements of one set K called the domain of the map to elements of another set V called the range of the map. Each element of the domain is often called a **key** and the corresponding element of the range is often called the **value**.

A map can be denoted by $f : K \to V$ or as a set of **key-value pairs** (k, v) denoted in the finite case by the set

$$f = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}.$$

of n pairs. We can also denote the pair (k, v) by v_k which looks like array notation except the subscripts do not need to be integers.

The keys themselves form the set $K = \{k_1, k_2, \ldots, k_n\}$ since no two keys can be the same. Since two or more keys can be associated with the same value, the values do not form a set, they form a collection.

13.9.1 Name-age example

As a simple example consider a set of names as the domain and the set of ages as the range. Then the following map associates names of people with their age.

$$\text{age} = \{(\text{Jane}, 12), (\text{Fred}, 10), (\text{Mary}, 15), (\text{Bob}, 10)\}.$$

Then, for example, using standard function notation, $\text{age}(\text{Fred}) = 10$ and $\text{age}(\text{Mary}) = 15$. A map can be visualized as a two-column table as shown in Figure 13.14. Here the keys go in the first

Name	Age
Jane	12
Fred	10
Mary	15
Bob	10

Figure 13.14: A two-column representation of the name-age map

column and the corresponding values go in the second column.

13.9.2 Basic map operations

The basic operations on a map are

add Add a new key-value pair to the map (a map should be resizable).

delete Remove a key-value pair given its key.

replace Replace the value in a key-value pair with a new value given its key.

search Search for (“look up”) the value associated with a given key.

The most important operation on a map is to be able to efficiently “look up” the value associated with a given key. A naive approach to this would be to use an array data structure to store the key-value pairs and, given a key, use a linear search to find the ordered pair containing this key and hence the value. This searching method would be $O(n)$.

A much better approach is to use a data structure called a hash table that uses a hash code to make lookup much more efficient than linear search. In fact look up is normally an $O(1)$ operation.

13.9.3 Hash tables and codes

We consider a very simple case of a hash table which is the implementation data structure for a map. In our case the keys and values are both integers. Suppose we have an array with indices 0 to 10 as shown in Figure 13.15 that can hold the key-value pairs. Here we assume that each array

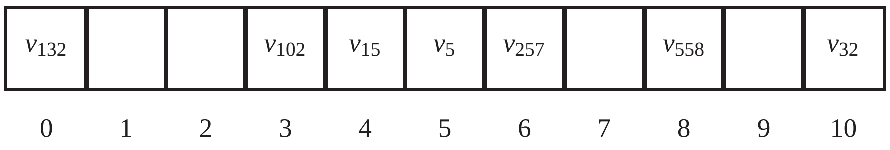


Figure 13.15: A simple hash table of size 11 using $h(k) = k$ mod 11. Here v_k is the value associated with key k.

location can hold one key-value pair and the notation v_k indicates that the value associated with key k is v_k and we assume that the values are non-negative integers. There is room for 11 pairs and some of them are shown in the figure. Empty array locations are unused.

For each key we need a function to transform the key into an array index which can then be used to obtain the value associated with this key.

In general the range of values (non-negative integers in this case) is much greater than the size of the array so we cannot just store the pair with key k in the location with index k. To be specific let us assume that each key k satisfies $0 \le k \le 1000$. What we need is a function $h(k)$ called a hash function that produces an integer hash code for each key k. This code can then be converted to an array index i in the range $0 \le i \le 10$ using $i = h(k)$ mod 11. We consider only the simplest case which is $h(k) = k$ so that the array index of key k is $i = k$ mod 11.

Suppose we start with an empty array and begin inserting pairs with keys 15, 558, 32, 132, 102, and 5. Then the corresponding array indices are 15 mod 11 $= 4$, 558 mod 11 $= 8$, 32 mod 11 $= 10$, 132 mod 11 $= 0$, 102 mod 11 $= 3$, and 5 mod 11 $= 5$, as shown in Figure 13.15.

No problems are encountered since all the remainders are different. However when we try to insert a pair with key 257 then 257 mod 11 $= 4$ and location 4 is already occupied by the pair v_{15} having key 15. This is inevitable as we insert new pairs since there are many more keys than array indices. This situation is called a **collision** and we need a **collision resolution policy** to decide where to store the pair. The simplest policy is to find the next highest empty location and store the pair there. In our example this means that pair v_{257}, which would have gone in the location with index 4, now goes in the location with index 6, as shown in Figure 13.15. In general we would assume that the array indices wrap around with index 0 following index 10. If there is no free location this means that the array is full and would need to be expanded by doubling its size for example.

13.10 The `Map<K,V>` interface

The JCF has a `Map<K,V>` interface that defines the basic operations on maps. This interface is parametrized with two generic types. The type `K` is the key type and the type `V` is the value type. They can be any object type. The methods in the `Map<K,V>` interface are shown in Figure 13.16. An interesting feature of this interface is that it contains an inner interface to represent the entries (pairs) in the map. Detailed descriptions of these operations are given in the Java API documentation which is summarized here.

- **`int size();`**

  Return the number of pairs (entries) currently stored in **`this`** map.

- **`boolean isEmpty();`**

  Return true if **`this`** map is empty (contains no entries).

- **`boolean containsKey(Object key);`**

  Return true if an entry with the given **`key`** is in **`this`** map.

- **`boolean containsValue(Object value);`**

  Return true if an entry with the given **`value`** is in **`this`** map.

- **`V get(Object key);`**

  Return the value associated with the given **`key`**. This is the “look up” operation. A return value of `null` either indicates that there is no entry with this key or there is an entry but its value is `null`.

- **`V put(K key, V value);`**

  Add a new pair (entry) to the map with given **`key`** and **`value`**. If the entry was already in **`this`** map then the old value is replaced by **`value`** and the old value is returned. Otherwise a new entry is added to **`this`** map and `null` is returned. This is an optional operation.

- **`V remove(Object key);`**

  If the entry with the given **`key`** is in **`this`** map then it is removed and its value is returned. Otherwise `null` is returned. This is an optional operation.

- **`void putAll(Map<? extends K, ? extends V> t);`**

  All the entries in the map **`t`** are put into **`this`** map. The types of the map **`t`** can be **`K`** and **`V`** or any types that extend or implement **`K`** and **`V`**. This is an optional operation.

```
public interface Map<K,V>
{
   // Query Operations
   int size();
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   V get(Object key);

   // Modification Operations
   V put(K key, V value); // optional
   V remove(Object key); // optional

   // Bulk Operations
   void putAll(Map<? extends K,? extends V> t); // optional
   void clear(); // optional

   interface Entry<K,V>
   {
      K getKey();
      V getValue();
      V setValue(V value); // optional
      boolean equals(Object obj);
      int hashCode();
   }

   // Views
   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K, V>> entrySet();

   // Comparison and hashing
   boolean equals(Object obj);
   int hashCode();
}
```

Figure 13.16: `Map` interface

- **`void clear();`**

  Remove all the entries from **`this`** map. The result is the empty map. This is an optional operation.

- **`interface Entry<K,V>`**

  This is an inner interface that defines a map entry (pair). To refer to such an entry use the type `Map.Entry<K,V>`.

  - **`K getKey();`**
    Return the key of **`this`** entry.
  - **`V getValue();`**
    Return the value of **`this`** entry.
  - **`V setValue(V value);`**
    Set a new value for **`this`** entry. This is an optional operation.
  - **`boolean equals(Object obj);`**
    Return true if **`obj`** is equal to **`this`** entry.
  - **`int hashCode()`**
    Return the hash code of **`this`** entry.

- **`Set<K> keySet();`**

  Return the keys in **`this`** map as a set.

- **`Collection<V> values();`**

  Return the values in **`this`** map as a collection.

- **`Set<Map.Entry<K,V>> entrySet();`**

  Return the entries of **`this`** map as a set of elements of type **`Map.Entry<K,V>`**.

- **`boolean equals(Object obj);`**

  Return true if **`obj`** is a map equal to **`this`** map.

- **`int hashCode()`**

  Return the hash code of **`this`** map.

13.11 Map implementations and examples

The JCF has several implementations of the `Map<K,V>` interface. We will consider three of them that are similar to the corresponding ones for sets: `HashMap<K,V>`, `LinkedHashMap<K,V>`, and `TreeMap<K,V>`.

```
public class HashMap<K,V> extends AbstractMap<K,V>
   implements Map<K,V>, Cloneable, Serializable
{
   public HashMap() {...}
   public HashMap(int initialCapacity) {...}
   public HashMap(Map<? extends K,? extends V> m) {...}
   public HashMap(int initialCapacity, float loadFactor) {...}

   public Object clone() {...}

   // Implementations of Map interface methods go here
}
```

Figure 13.17: The `HashMap<K,V>` class

```
public class LinkedHashMap<E> extends HashMap<K,V>
   implements Map<K,V>, Cloneable, Serializable
{
   public LinkedHashMap() {...}
   public LinkedHashMap(int initialCapacity) {...}
   public LinkedHashMap(int initialCapacity, float loadFactor) {...}
   public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {...}
   public LinkedHashMap(Map<? extends K,? extends V> m) {...}

   public Object clone() {...}

   // Implementations of Map interface methods go here
   // Other methods go here
}
```

Figure 13.18: The `LinkedHashMap<K,V>` class

13.11.1 `HashMap<K,V>` implementation of `Map<K,V>`

The `HashMap<K,V>` implementation is the fastest but it does not maintain any order to the entries in the map. A class summary is shown in Figure 13.17.

There are four constructors. The first constructor with no arguments constructs an empty map with a default initial capacity of 16 elements. The second constructor specifies a given initial capacity. The third one is called a conversion constructor and can be used as a copy constructor. We will not use the fourth constructor. It is used to optimize the hash table implementation.

13.11.2 `LinkedHashMap<K,V>` implementation of `Map<K,V>`

The `LinkedHashMap<K,V>` implementation maintains the order in which keys are added to the map. A class summary is shown in Figure 13.18.

```
public class TreeMap<K,V> extends AbstractMap<K,V>
   implements SortedMap<K,V>, Cloneable, Serializable
{
   public TreeMap() {...}
   public TreeMap(Comparator<? super K> c) {...}
   public TreeMap(Map<? extends K,? extends V> m) {...}
   public TreeMap(SortedMap<K,? extends V> m) {...}

   public Object clone() {...}

   // implementations of SortedMap interface go here
   // SortedMap extends the Map interface
}
```

Figure 13.19: The `TreeMap<K,V>` class

13.11.3 `TreeMap<K,V>` implementation of `Map<K,V>`

The `TreeMap<K,V>` implementation provides a sorted order based on the natural ordering of the keys as given by the `Comparable<K>` interface implemented by `K`. A class summary is shown in Figure 13.19. The `SortedMap<K,V>` interface extends the `Map<K,V>` interface to provide extra methods related to the sort order (See Java API documentation).

13.11.4 Simple map examples

Here we give some simple examples to illustrate map operations using the name-age example.

■ EXAMPLE 13.19 **(Constructing a name-age map)** The statements

```
Map<String,Integer> age = new HashMap<String,Integer>();
age.put("Jane", 12);
age.put("Fred", 10);
age.put("Mary", 15);
age.put("Bob", 10);
System.out.println(age);
```

create the name-age map shown in Figure 13.15 using autoboxing from `int` to `Integer`. The no-arg constructor uses a default size of 16 entries for the map. The output is

```
{Bob=10, Jane=12, Fred=10, Mary=15}
```

and shows that the insertion order is not preserved by the `HashMap` implementation. If you change the implementation to `LinkedHashMap` then the output is

```
{Jane=12, Fred=10, Mary=15, Bob=10}
```

which is in the order of insertion into the map. Finally, if you change the implementation to `TreeMap` then the output is

```
{Bob=10, Fred=10, Jane=12, Mary=15}
```

which is sorted in increasing order of the names (keys). ■

■ EXAMPLE 13.20 **(Finding the age of a given person)** The statements

```
String name = "Mary";
int a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

return the age of Mary. ■

■ EXAMPLE 13.21 **(Using get if name is not in the map)** The statements

```
String name = "Gord";
int a = age.get(name);
```

throw a `NullPointerException`. Since Gord is not in the map `get` returns `null` which cannot be unboxed to an `int` so the exception is thrown. This only happens with the primitive types. Without the auto unboxing the statements

```
String name = "Gord";
Integer a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

return a `null` value for `a` and no exception is thrown. ■

■ EXAMPLE 13.22 **(Checking if a map contains a key)** The statements

```
String name = "Jill";
if (age.containsKey(name))
   System.out.println(name + " was found");
else
   System.out.println(name + " was not found");
```

show that Jill was not found in the map. ■

■ EXAMPLE 13.23 **(Update a value given its key)** The statements

```
String name = "Fred";
age.put(name, 15);
System.out.println("New age of " + name + " is " + age.get(name));
```

update the age of Fred from 10 to 15 and display it. The statements

```
String name = "Fred";
int currentAge = age.get(name);
age.put(name, currentAge + 1);
System.out.println("New age of " + name + " is " + age.get(name));
```

add 1 year to Fred's age and display it. ■

■ EXAMPLE 13.24 **(Deleting an entry given its key)** The statements

```
String name = "Fred";
if (age.containsKey(name))
   age.remove(name);
System.out.println(age);
```

delete Fred from the map and display the resulting map

```
{Bob=10, Jane=12, Mary=15}
```

which shows that Fred is no longer an entry in the map ■

■ EXAMPLE 13.25 **(Iterating over the keys of a map)** To get an iterator over the keys in a map we first get the keys as a set and then ask this set for an iterator. The statements

```
Set<String> keys = age.keySet();
Iterator<String> iter = keys.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   int a = age.get(name);
   System.out.println(name + " -> " + a);
}
```

use the iterator to display the name-age pairs using an "arrow" notation, one per line. ■

■ EXAMPLE 13.26 **(Iterating over the keys using a for-each loop)** The statements

```
for (String name : age.keySet())
{
   System.out.println(name + " -> " + age.get(name));
}
```

use the for-each loop to display the name-age pairs using an "arrow" notation, one per line. ■

■ EXAMPLE 13.27 **(Use the for-each loop to compute average age)** The statements

```
Set<String> keys = age.keySet();
double sum = 0.0;
for (String name : keys)
{
   sum += age.get(name);
}
System.out.println("Average age is " + sum / keys.size());
```

compute the average age. The `size` method is used to find the number of keys in the map. ■

■ EXAMPLE 13.28 **(Use an iterator and the `Map.Entry` interface)** The statements

```
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();
while (iter.hasNext())
{
   Map.Entry<String,Integer> entry = iter.next();
   System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

iterate over the map entries. First we get the `entries` set of type `Map.Entry<String,Integer>` using the inner interface of the `Map<String,Integer>` interface. Then we ask it for an iterator over the entries. Each entry has `getKey()` and `getValue()` methods. The loop displays the entries using arrow notation.

The for-each loop

```
for (Map.Entry<String,Integer> entry : age.entrySet())
{
   System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

can be used as long as the mutable iterator operations are not required. ■

■ EXAMPLE 13.29 **(Adding 1 year to all the ages)** The statements

```
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();
while (iter.hasNext())
{
   Map.Entry<String,Integer> entry = iter.next();
   entry.setValue(entry.getValue() + 1);
}
System.out.println(entries);
```

use the `entrySet()` iterator to add 1 to all the ages. ■

13.11.5 Hours worked example

As a useful example of a map suppose we have a file called `hours.txt` whose lines contain a person's name and the number of hours they have worked. An example might be

```
Fred:10
Gord:20
Fred:30
Mary:15
Gord:13
Mary:4
Mary:6
```

There can be more than one entry per person and we want to display the total hours worked by each person in the format

```
Fred -> 40.0
Mary -> 25.0
Gord -> 33.0
```

indicating that Fred has worked 40 hours (10 + 30), Gord has worked 33 hours (20 + 13), and Mary has worked 25 hours (15 + 4 + 6).

We can produce this list by reading the file into a map with the names as keys and the hours worked as the values. Each time we read a line we check if the name is already in the map. If it is not we create a new entry, and if it is already in the map we update the number of hours by adding the new value.

Before reading the file we create the following map:

```
Map<String,Double> map = new HashMap<String,Double>();
```

If you want the names to be ordered alphabetically then replace `HashMap` by `TreeMap`.

Then if `name` and `hours` are the values read from the file the map is updated using the statements

```
if (map.containsKey(name)) // update hours worked
{
   double currentHours = map.get(name);
   map.put(name, currentHours + hours);
}
else // new entry
{
   map.put(name, hours);
}
```

To read the lines of the file we can use the `split` method in the `String` class, so if `line` is a line read from the file then

```
String[] s = line.split(":");
```

will read the name and hour values as strings into `s[0]` and `s[1]`, using colon as the delimiter. Here is the complete program.

Class `HoursWorked`

`book-project/chapter13/maps`

```
package chapter13.maps;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```
/**
 * A map example: file contains names and hours worked in the format
 * name:hours
 * A person may appear several times in the file and we want to
 * determine the total hours each person has worked.
 *
 * We read this file a line at a time and separate the name and hours.
 * The name is used as the key in a hash table and the hours is the
 * value if the key is new else the hours are updated. The result
 * is a map containing the total hours worked for each person.
 *
 * If a TreeMap is used instead of a HashMap the names will be
 * ordered in increasing alphabetic order.
 */
public class HoursWorked
{
   private static final File IN_FILE = new File("files/hours.txt");

   public void processFile() throws IOException
   {
      Map<String,Double> map =
         new HashMap<String,Double>();
      BufferedReader in =
         new BufferedReader(new FileReader(IN_FILE));
      String line;

      while ( (line = in.readLine()) != null)
      {
         // Each line of the file contains a name and a number
         // of hours worked separated by a colon which can be
         // preceded by zero or more spaces.

         String[] s = line.split(":");
         String name = s[0].trim();
         double hours = Double.parseDouble(s[1].trim());

         // Echo for checking

         System.out.println(name + ":" + hours);

         // put entries in map and update hours

         if (map.containsKey(name)) // update hours worked
         {
            double currentHours = map.get(name);
            map.put(name, currentHours + hours);
         }
         else // new entry
         {
            map.put(name, hours);
         }
      }
      in.close();

      // Display the map, one entry per line

      System.out.println("Map is");
      for (String name : map.keySet())
      {
         double hours = map.get(name);
         System.out.println(name + " -> " + hours);
      }
   }

   public static void main(String[] args) throws IOException
   {
      HoursWorked tester = new HoursWorked();
      tester.processFile();
   }
}
```

13.11.6 Favorites map with maps as values

We now do an example of a map whose values are also maps. This example is extended in the end of chapter exercises.

In our case we want a map structure that can record the favorite song, food, golfer, etc, associated with each person. Thus, the key-value pairs of the primary map are names and references to favorite maps. The key-value pairs of each favorite map are the category names, such as food, song and golfer, and the values are the preferences.

Using set theory notation an example of such a map of maps is

$$\begin{aligned}
\text{favorites} &= \{(\text{Bob}, f_1), (\text{Fred}, f_2), (\text{Gord}, f_3)\} \\
f_1 &= \{(\text{food}, \text{salad}), (\text{golfer}, \text{Vijay Singh}), (\text{song}, \text{White Wedding})\} \\
f_2 &= \{(\text{food}, \text{steak}), (\text{golfer}, \text{Tiger Woods}), (\text{song}, \text{Satisfaction})\} \\
f_3 &= \{(\text{food}, \text{spaghetti}), (\text{golfer}, \text{Phil Mickelson}), (\text{song}, \text{Money})\}
\end{aligned}$$

This example is also shown using tables in Figure 13.20. It is easy to construct these maps in Java. The favorites map is given by

```
Map<String,Map<String,String>> favorites =
   new HashMap<String, Map<String,String>>();
```

which is a map from strings to maps from strings to strings. Now the favorite maps are given by

```
Map<String,String> f1 = new HashMap<String,String>();
f1.put("golfer", "Vijay Singh");
f1.put("song", "White Wedding");
f1.put("food", "salad");

Map<String,String> f2 = new HashMap<String,String>();
```

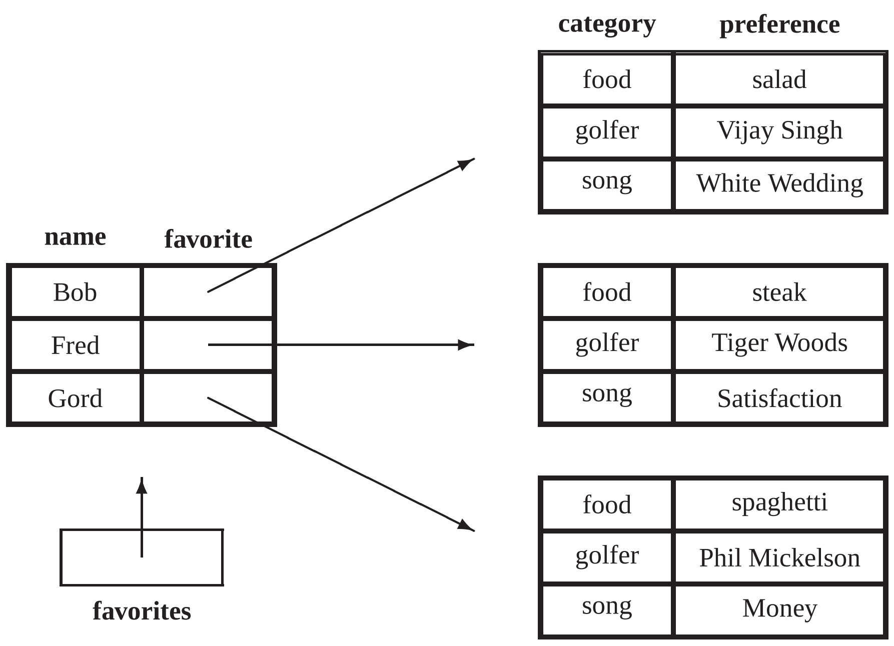


Figure 13.20: A map of maps. The keys of the first map are names. The values are favorite maps whose keys are the categories and values are the preferences.

```
f2.put("golfer", "Tiger Woods");
f2.put("song", "Satisfaction");
f2.put("food", "steak");

Map<String,String> f3 = new HashMap<String,String>();
f3.put("golfer", "Phil Mickelson");
f3.put("song", "Money");
f3.put("food", "spaghetti");
```

Finally we associate these maps as values of the favorites map:

```
favorites.put("Bob", f1);
favorites.put("Fred", f2);
favorites.put("Gord", f3);
```

It is easy to perform operations on this map. For example, to display Fred's favorite map use

```
System.out.println(favorites.get("Fred"));
```

To display Bob's favorite golfer use

```
System.out.println(favorites.get("Fred").get("golfer"));
```

To change Fred's favorite food to chicken use

```
favorites.get("Fred").put("food", "chicken");
```

■ EXAMPLE 13.30 **(For-each loop for favorites map)** The statements

```
for (String name : favorites.keySet())
{
   System.out.println(name);
   System.out.println(favorites.get(name));
}
```

produce the output

```
Bob
{golfer=Vijay Singh, food=salad, song=White Wedding}
Fred
{golfer=Tiger Woods, food=steak, song=Satisfaction}
Gord
{golfer=Phil Mickelson, food=spaghetti, song=Money}
```

which show the favorite maps one per line. To obtain an alphabetical order replace the `HashMap` implementation by `TreeMap`. ■

■ EXAMPLE 13.31 **(Nested for-each loops for favorites map)** The statements

```
for (String name : favorites.keySet())
{
   System.out.println("favorites for " + name + ":");
   Map<String,String> favorite = favorites.get(name);
   for (String  category : favorite.keySet())
   {
      String preference = favorite.get(category);
      System.out.println("   " + category + ": " + preference);
   }
}
```

produce the display

```
favorites for Bob:
   food: salad
   golfer: Vijay Singh
   song: White Wedding
favorites for Fred:
   food: steak
   golfer: Tiger Woods
   song: Satisfaction
favorites for Gord:
   food: spaghetti
   golfer: Phil Mickelson
   song: Money
```

using nested for-each loops to iterate over the maps. The outer loop iterates over each person and the inner loop iterates over all categories in each favorite map. ■

13.12 Recursion examples using maps

Consider a sequence $[s_m, s_{m+1}, s_{m+2}, \ldots, s_n, s_{n+1}, \ldots]$ with starting index m which is often taken to be 0. Such sequences are often defined by recurrence relations of the form $s_n = f(s_{n-1})$, which is a first order recurrence relation since the calculation of s_n depends on the previous term in the sequence, or of the form $s_n = f(s_{n-1}, s_{n-2})$, which is a second-order recurrence relation since the calculation of s_n depends on the previous two terms of the sequence. As a simple example, the recurrence relation $s_n = ns_{n-1}$ with $s_0 = 1$ can be solved to get $s_n = n!$.

Here we consider two recurrence relations, the Fibonacci sequence and the Q-sequence.

13.12.1 The Fibonacci sequence

An important second-order sequence is the Fibonacci sequence defined recursively by

$$F_n = F_{n-1} + F_{n-2}, \text{ where } F_0 = F_1 = 1.$$

There is a closed form expression for the general term F_n but it is not useful for the calculation of terms in the sequence. An efficient non-recursive method is easily written to calculate the terms in the sequence and the following recursive method can also be used

```
public long fib(int n)
{
   if (n == 0 || n == 1)
      return 1L;
   else
      return fib(n-1) + fib(n-2);
}
```

This method is very inefficient because each term is calculated many times. For example, in the calculation of f_{30} the term f_{10} is calculated recursively $10,946$ times.

We can avoid this duplication by a technique called memoization. In our case this means that we can use a map to remember the terms as they are calculated. When we calculate a term for the first time we store it in a map of type `Map<Integer,Long>`. Then whenever this term is needed again we simply look up its value in the map. Here is a class that calculates Fibonacci numbers using a map:

Class `Fibonacci`

book-project/chapter13/maps

```
package chapter13.maps;
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class Fibonacci
{
   Map<Integer,Long> m;
   public void calculate()
   {
      // Create map and initialize it
      // for fib(0)= 1 and fib(1)= 1

      m = new HashMap<Integer,Long>();
      m.put(0,1L);
      m.put(1,1L);

      Scanner input = new Scanner(System.in);
      System.out.println("Enter n");
      int n = input.nextInt();

      long startTime = System.nanoTime();
      System.out.println(fib(n));
      long time = System.nanoTime() - startTime;

      double seconds = (double) time * 1e-9;
      System.out.println(seconds);
   }

   public long fib(int n)
   {
      if (! m.containsKey(n))
         m.put(n, fib(n-1) + fib(n-2));
      return m.get(n);
   }

   public static void main(String[] args)
   {
      new Fibonacci().calculate();
   }
}
```

Note that before calling `fib` we construct the map and initialize it by putting the entries for $F_0 = 1$ and $F_1 = 1$ into it.

The `fib` method first checks to see if the term F_n is in the map. If it isn't the recursive formula is used to calculate it and put it in the map, otherwise it is looked up in the map and returned.

We have included statements that determine the time in seconds taken to compute a Fibonacci number. A similar class could be written for the recursive version without using a map. Of course the results depend on the particular computer. In one test the calculation of F_{46} took 53.8 seconds without using a map and 3.43×10^{-4} seconds using a map.

13.12.2 The Q-sequence

As another more complicated example which doesn't have a simple non-recursive algorithm consider the sequence

$$Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), \text{ where } Q(1) = 1, Q(2) = 1$$

where we use the more readable function notation $Q(n) = Q_n$. The following recursive method can be used to compute the terms in the sequence.

```
public int q(int n)
{
   if (n <= 2)
      return 1;
   else
      return q(n - q(n-1)) + q(n - q(n-2));
   }
```

The following class uses a map to calculate the terms:

Class `QSequence`

`book-project/chapter13/maps`

```
package chapter13.maps;
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class QSequence
{
   Map<Integer,Integer> m;

   public void calculate()
   {
      // Create map and initialize it
      // for q(1) = 1 and q(2) = 1

      m = new HashMap<Integer,Integer>();
      m.put(1,1);
      m.put(2,1);

      Scanner input = new Scanner(System.in);
      System.out.println("Enter n");
      int n = input.nextInt();
      long startTime = System.nanoTime();
      System.out.println(q(n));
      long time = System.nanoTime() - startTime;
      double seconds = (double) time * 1e-9;
      System.out.println(seconds);
   }

   public int q(int n)
   {
      if (! m.containsKey(n))
         m.put(n, q(n - q(n-1)) + q(n - q(n-2)));
      return m.get(n);
   }
   public static void main(String[] args)
   {
      new QSequence().calculate();
   }
}
```

In one test the calculation of $Q(45)$ took 75.8 seconds without using a map and 4.35×10^{-4} seconds using a map.

13.13 `Collections` utility class

The `Collections` class is like the `Math` class: it is a set of useful static methods such as sorting and searching for operating on sets, lists, and maps in the JCF. There are 50 methods in this class and we summarize only a few. For a complete description see the Java API documentation.

- **`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key))`**

  Search **`list`** of type **`T`** for the given **`key`**. The list must be in the order specified by the `Comparable` interface implemented by the list. Returns the zero-based index where **`key`** was found or (`-index - 1`) where `index` is the location where **`key`** could be inserted.

- **`static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`**

  Like the above version of **`binarySearch`** except using the specified implementation of **`Comparator`** to define the order. (**`list`** does not need to implement `Comparable` in this version).

- **`static <T extends Comparable<? super T>> void sort(List<T> list)`**

  Sort the given **`list`** into increasing order using the implementation of the `Comparable` interface provided by **`list`**.

- **`static <T> void sort(List<T> list, Comparator<? super T> c)`**

  Like the above version of **`sort`** except using the specified implementation of **`Comparator`** to define the order. (**`list`** does not need to implement `Comparable` in this version).

There is also an `Arrays` class in `java.util` that provides a similar set of static methods that operate on arrays instead of collections.

13.13.1 Book list sorting example

In this example we consider two ways to use the `sort` method in the `Collections` class to sort a list of `Book` objects (see page 760).

The `Book` class implements `Comparable<Book>` which defines the natural order to be increasing alphabetical order by book title. This means that we can sort a book list in this order simply by using

```
Collections.sort(list);
```

where `list` is a list of books.

If we want to use an order other than the natural order it is necessary to write a class that implements the `Comparator<Book>` interface. For example, if we want to sort in increasing alphabetic order by author then following class can be used

Class `BookComparator`

`book-project/chapter13/lists`

```
package chapter13.lists;
import java.util.Comparator;

public class BookComparator implements Comparator<Book>
{
   /**
    * Compare this book to another book using the author.
    * @param b1 the first book
    * @param b2 the second book
    * @return negative, zero, positive results
    */
   public int compare(Book b1, Book b2)
   {
      return b1.getAuthor().compareTo(b2.getAuthor());
   }
}
```

Now we can use the statement

```
Collections.sort(list, new BookComparator());
```

to sort by author. Here is a class that illustrates these two sorting methods:

Class `SortBookList`

`book-project/chapter13/lists`

```
package chapter13.lists;
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class SortBookList
{
   public void processBookList()
   {
      // A simple list of books

      List<Book> list = new ArrayList<Book>();
      list.add(new Book("Dead Souls", "Ian Rankin", 25.95 ,10));
      list.add(new Book("Stranger House", "Reginald Hill", 29.50 ,0));
      list.add(new Book("Not Safe After Dark", "Peter Robinson", 32.99 ,10));
      list.add(new Book("Original Sin", "P. D. James", 39.95 ,0));
      list.add(new Book("Fleshmarket Close", "Ian Rankin", 25.00 ,0));

      // Sort using the sort method in the Collections class
      // The order uses titles (Book implements Comparable)

      Collections.sort(list);
      System.out.println("List sorted by title:");
      displayList(list);

      // Now use a Comparator the sorts using the author

      Collections.sort(list, new BookComparator());
      System.out.println("List sorted by author:");
      displayList(list);
   }

   public static <E> void displayList(List<E> list)
   {
      for (E element : list)
         System.out.println(element);
   }

   public static void main(String[] args)
   {
      SortBookList books = new SortBookList();
      books.processBookList();
   }
}
```

The output is

```
      List sorted by title:
      Book[Dead Souls,Ian Rankin,25.95,10]
      Book[Fleshmarket Close,Ian Rankin,25.0,0]
      Book[Not Safe After Dark,Peter Robinson,32.99,10]
      Book[Original Sin,P. D. James,39.95,0]
      Book[Stranger House,Reginald Hill,29.5,0]
      List sorted by author:
      Book[Dead Souls,Ian Rankin,25.95,10]
      Book[Fleshmarket Close,Ian Rankin,25.0,0]
      Book[Original Sin,P. D. James,39.95,0]
      Book[Not Safe After Dark,Peter Robinson,32.99,10]
      Book[Stranger House,Reginald Hill,29.5,0]
```

13.14 Programming exercises

► Exercise 13.1 (A random `remove` method)
Modify the `Bag<E>` interface on page 729 by adding a random `remove` method with prototype

```
E remove();
```

that removes a random element from this bag and returns it. If the bag is empty then `null` is returned. Write the method implementation (it will be the same for both `FixedBag<E>` and `DynamicBag<E>`). You can use the `Random` class in `java.util` that has a `nextInt` method.

► Exercise 13.2 (An indexed `add` method)
For the `Array<E>` interface add a method with prototype

```
void add(int k, E element);
```

that adds the given `element` at index `k`. The method should throw `IndexOutOfBoundsException` if `k < 0` or `k > size()`.

The element originally at position `k` and all following elements need to be moved up one place to create a place for the new element. The special case when `k` has the value `size()` corresponds to adding the element at the end of the array.

Write the implementation of this method for the `DynamicArray<E>` implementation of the `Array<E>` interface.

► Exercise 13.3 (An indexed `remove` method)
For the `Array<E>` interface add a method with prototype

```
E remove(int k);
```

that removes the element at index `k` by shifting all following elements down one place. The element removed is returned.

Write the implementation of this method for the `DynamicArray<E>` implementation of the `Array<E>` interface.

► Exercise 13.4 (An `indexOf` method)
Modify the `Array<E>` interface on page 738 to include an `indexOf` method with prototype

```
int indexOf(E element);
```

that returns the index of the first occurrence of the given element `E` or -1 if the element is not found. Write the method implementation for `DynamicArray<E>`.

► Exercise 13.5 (Generating random sets of numbers)
Using the idea in Example 13.12, write a method with prototype

```
Set<Integer> randomSet(int n, int a, int b, Random random);
```

that returns a set of `n` integers randomly chosen in the range `a` to `b` inclusive using the `Random` class in `java.util`.

▶ **Exercise 13.6 (Generating Lotto 649 numbers using sets)**
Using Exercise 13.5 write a class called `Lotto649` that generates *n* sets of 6 numbers in the range 1 to 49 and displays them.

▶ **Exercise 13.7 (Generating Lotto 649 numbers without using sets)**
Without using the JCF write a class called `Lotto649NoSets` that generates n sets of 6 numbers in the range 1 to 49 and displays them.

▶ **Exercise 13.8 (Password generator using sets)**
We want to generate a set of unique passwords. Each password is made from the lower and upper case letters and the digits and has a specified length. Write a class to do this. The input is the number of passwords in the set and the number of characters in each password (same for all passwords in the set). Some sample output is

```
d6rIH hX9Av Ki4SK wDAWx olWhW TVU7Y hGDSw VZecI ga7Sy 0DEij
7aDws T0urW MMjk9 JDAHZ vRb1x lGz3q ibiuE H7nbF CB6zY EGzuX
Dhiou mLtkI Eud22 wNbVo iIhLZ Zc73V taPFL wPJGZ nOy9x DPx9F
leJv3 KhmqQ y23g0 ey3Kr VQvq1
```

corresponding to a set size of 35 with 5 characters in each password. Display 10 passwords per line except possibly for the last line.

▶ **Exercise 13.9 (Printing a collection one element per line)**
The standard `toString` method creates a string which, when displayed, is all on one line. Write a static polymorphic method with prototype

```
public static <E> void printCollection(Collection<E> c)
```

that prints the elements one per line.

▶ **Exercise 13.10 (Removing duplicate words)**
Write a class similar to `RemoveDuplicateWords` on Page 753 and called `UniqueWords` that creates two sets. The first is a set of unique words as defined in the `RemoveDuplicateWords` class, and the second is a set of duplicate words (words that appeared more than once in the input file). From these sets create a set of words that did not have any duplicates in the input. For example for the input

```
a b c d a b e
```

the output should be

```
3 unique words found:
[c,d,e]
2 duplicate words found:
[a,b]
```

▶ **Exercise 13.11 (Adapter class version of the Bag ADT)**
Write an adapter class implementation `ArrayBag<E>` of the `Bag<E>` interface on page 729 that adapts an `ArrayList<E>` object. The adapter class has the following structure

```
import java.util.ArrayList;

/**
 * An adapter class implementation of Bag<E>
 */
public class Bag<E>
{
   // This is an adapter class version of the
   // bag ADT that uses an ArrayList

   private ArrayList<E> bag;

   public Bag() {...}
   public Bag(int initialCapacity) {...}

   /**
    * Copy constructor.
    * @param b the bag to copy
    */
   public Bag(Bag<E> b) {...}

   public int size() {...}
   public boolean isEmpty() {...}

   public boolean add(E element) {...}
   public boolean remove(E element) {...}
   public boolean contains(E element) {...}

   public String toString()
   {
      return "Bag" + bag.toString();
   }
}
```

Here all methods are implemented using the `bag` object instance data field of type `ArrayList<E>`.

► Exercise 13.12 (Memory tester game)
Write a class called `MemoryTester` that uses the `DynamicBag<Integer>` class and the algorithm shown in Figure 13.21.

Here is some typical output assuming that there are 5 numbers to guess and the numbers are in the range 1 to 10

```
Bag[9,9,8,5,4]
Enter guesses for the 5 numbers in range 1 to 10
9 9 7 5 3
You have 3 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
```

```
ALGORITHM MemoryGame()
Make a bag that can hold 5 integers.
Generate 5 random integers in the range 1 to 10
    and add them to the bag.
LOOP
    Make a copy of the original bag
    Ask user for 5 guesses of numbers in the bag
        and remove the guesses from the bag copy if possible.
    IF bag copy is now empty THEN
        EXIT LOOP
    END IF
    Determine how many guesses are correct.
    Tell user how many guesses are correct.
END LOOP
Congratulate user on winning the game.
```

Figure 13.21: Memory game algorithm

```
9 9 8 5 5
You have 4 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
8 8 7 4 3
You have 2 guesses correct
Enter guesses for the 5 numbers in range 1 to 10
9 9 8 5 4
Congratulations all guesses are correct
```

Here the first line actually shows the answer so that you can check your class. When it is working you can remove this display.

► **Exercise 13.13 (Cities and Countries map)**
We start with the following text file `cities.txt`

```
Toronto:Canada
Chicago:USA
Frankfort:Germany
Sudbury:Canada
Venice:Italy
Acapulco:Mexico
Berlin:Germany
Barcelona:Spain
Los Angeles:USA
Vancouver:Canada
Rome:Italy
```

```
Miami:USA
London:UK
Mexico City:Mexico
Madrid:Spain
Florence:Italy
```

that is a list of cities and their countries. In general each country can appear several times.

We want to read this file a line at a time and produce a file `countries.txt` having the form

```
Canada -> [Sudbury, Toronto, Vancouver]
Germany -> [Berlin, Frankfort]
Italy -> [Florence, Rome, Venice]
Mexico -> [Acapulco, Mexico City]
Spain -> [Barcelona, Madrid]
UK -> [London]
USA -> [Chicago, Los Angeles, Miami]
```

The output has the form of a map from `String` to `List<String>`:

```
Map<String,List<String>> map = new TreeMap<String,List<String>>();
```

which will arrange the countries in sorted order.

Write a class called `Cities` to solve this problem. You can use an `ArrayList<String>` for each list. See Section 13.11.5 for a simpler example, To sort the lists for each country, before using `PrintWriter` to write the results to a file, use the static `sort` method in the `Collections` class.

► Exercise 13.14 (Another version of the cities and countries map)
Write a version of the `Cities` class from the previous exercise called `Cities2` that produces the same output but expects its input in the compact form

```
Toronto:Canada, Chicago:USA, Frankfort:Germany, Sudbury:Canada
Venice:Italy, Acapulco:Mexico, Berlin:Germany
Barcelona:Spain, Los Angeles:USA, Vancouver:Canada
Rome:Italy, Miami:USA, London:UK
Mexico City:Mexico, Madrid:Spain, Florence:Italy
```

that permits multiple entries per line in the input file separated by commas (use a nested loop with the outer loop using `split(",")` in the outer loop and `split(":")` in the inner loop.

► Exercise 13.15 (Favorites map using data file)
We want to read a text file `favorites.txt` such as

```
Fred:golfer:Tiger Woods
Bob:food:salad
Fred:food:steak
Bob:song:White Wedding
Gord:golfer:Phil Mickelson
Fred:song:Satisfaction
Bob:golfer:Vijay Singh
Gord:song:Money
Gord:food:spaghetti
```

and produce the display shown in Example 13.31 which also corresponds to the favorites map given in Figure 13.20.

Write a class called `Favorites` that does the processing using the map structure of Section 13.11.6. Also see Example 13.30 and Example 13.31.

► Exercise 13.16 (ArrayList version of an address book)
The purpose of this exercise is to write a GUI version of an address book program that uses an `ArrayList` to hold the address book entries. Each entry is an object from an inner class called `AddressBookEntry`. An entry is really two strings, one called the `key` for the name of the person and another called the `value` representing the address information.

Now we need to write a class called `AddressBook` that manages the address book. This class will be used by the GUI class `AddressBookGUI`. The `AddressBook` class has the following structure which you must complete as indicated by the `TODO` lines.

```
public class AddressBook
{
   private List<AddressBookEntry> list; // the list of address book entries
   private String fileName; // name of file containing the list
   private String fileStatus; // Status or error message or empty

   /**
    * Construct an empty address book with a given initial
    * size. No attempt is made to read an address book from
    * a binary object file so this constructor is mainly
    * used for debugging.
    * @param initialSize the initial address book size
    */
   public AddressBook(int initialSize)
   {
      list = new ArrayList<AddressBookEntry>(initialSize);
      fileName = "";
      fileStatus = "";
   }

   /**
    * Make an address book from the data in a binary object file,
    * if the file exists, else construct a new address book. Errors
    * are recorded as strings that can be retrieved using the
    * fileStatus() method.
    */
   public AddressBook(String inFileName)
   {
      fileName = inFileName;
      read();
   }

   /**
```

```
 * Read the address book from a binary object file. If
 * a binary object file does not exist then create a new
 * address book.
 * Errors are recorded as strings that can be retrieved using the
 * fileError() method.
 */
public void read()
{
   // If no error then the string is empty
   fileStatus = "";

   ObjectInputStream in = null;
   try
   {
      in = new ObjectInputStream(new FileInputStream(fileName));
      list = (List<AddressBookEntry>) in.readObject();
      fileStatus = "Address book file has been loaded";
   }
   catch (FileNotFoundException e)
   {
      // make a new address book if input file not found.
      list = new ArrayList<AddressBookEntry>();
      fileStatus = "New address book list has been created";
   }
   catch (ClassNotFoundException e)
   {
      fileStatus = "Invalid address book file";
   }
   catch (IOException e)
   {
      fileStatus = "Unknown error reading address book file";
   }
   finally // make sure the file was closed
   {
      try
      {
         if (in != null) in.close();
      }
      catch (IOException e)
      {
         fileStatus = "Unknown error closing address book file";
      }
   }
}

/**
```

```
 * Write the address book as a binary object file.
 * Errors are recorded as strings that can be retrieved using the
 * fileError() method.
 */
public void write()
{
   fileStatus = "Address book has been saved in file";
   ObjectOutputStream out = null;
   try
   {
      out = new ObjectOutputStream(new FileOutputStream(fileName));
      out.writeObject(list);
   }
   catch (FileNotFoundException e)
   {
      fileStatus = "Address book file not found";
   }
   catch (IOException e)
   {
      fileStatus = "Unknown error writing address book file";
   }
   finally
   {
      try
      {
         if (out != null) out.close();
      }
      catch (IOException e)
      {
         fileStatus = "Unknown error closing output file";
      }
   }
}

/**
 * Return file error or status string after a file operation.
 * @return the status string.
 */
public String fileStatus()
{
   return fileStatus;
}

/**
 * Return number of entries in address book.
 * @return number of entries in address book
```

```
         */
        public int size()
        {
           return list.size();
        }

        /**
         * Return the value associated with a given key.
         * @param key the key to find
         * @return value of key found else null
         */
        public String get(String key)
        {
           // TODO
        }

        /**
         * Add a new entry to the address book.
         * If the entry already exists then it is an update operation
         * so the value of the entry for this key is updated.
         * @param key the key of entry to add or update
         * @param value new value for the entry
         */
        public void add(String key, String value)
        {
           // TODO
        }

        /**
         * Delete an entry from address book given its key.
         * @param key the key of the entry
         * @return true if entry was deleted
         * else false if entry did not exist.
         */
        public boolean delete(String key)
        {
           // TODO
        }

        /**
         * Return a string representation of this list.
         * @return a string representation of this list.
         */
        public String toString()
        {
           // TODO
```

```
   }

   // -------- inner class for address book entries ------------

   /**
    * An object of this class is an entry in an address book database.
    * Each entry is a key-value pair. The keys and values are strings.
    */
   private static class AddressBookEntry implements java.io.Serializable
   {
      private static final long serialVersionUID = 1L;
      private String key;
      private String value;

      /**
       * Construct an entry given a key and a value.
       * @param key the key for the entry.
       * @param value the value associated with the key.
       */
      public AddressBookEntry(String key, String value)
      {
         this.key = key;
         this.value = value;
      }

      /**
       * Return the key for this entry.
       * @return Return the key for this entry.
       */
      public String getKey()
      {
         return key;
      }

      /**
       * Return the value associated with this key
       * @return Return the value associated with this key.
       */
      public String getValue()
      {
         return value;
      }

      /**
       * Test this object for equality with obj
       * @param obj the object to test with this object
```

```
              * @return true if the two objects have the same keys else false
              */
             public boolean equals(Object obj)
             {
                if (obj == null) return false;
                if (! getClass().equals(obj.getClass())) return false;
                AddressBookEntry entry = (AddressBookEntry) obj;
                return key.equals(entry.key);
             }

             /**
              * Define a string representation of this object.
              * @return Return the string representation of this object.
              */
             public String toString()
             {
                return "AddressBookEntry[" + key + ", " + value + "]";
             }
          }
       }
```

Now write the `AddressBookGUI` class that uses the `AddressBook` class. This class can have a `JTextField` for the key, and a `JTextArea` for the value. Another `JTextArea` can be used to display output and status information and `JButton` objects can be used for ”Save”, ”Search”, ”Delete”, ”Add”, and ”Display All” operations.

► **Exercise 13.17 (Map version of an address book)**
Repeat the previous exercise using a map of the type `Map<String,String>` instead of a list to hold the address book entries. Now there is no need for the inner `AddressBookEntry` class. The GUI class will be the same as in the previous exercise, only the `AddressBook` class will change.

► **Exercise 13.18 (Map version of a telephone directory)**
Write a GUI version of telephone directory that uses a sorted map.