

OBJECT-ORIENTED PROGRAMMING IN JAVA™

Richard L. Halterman

13 March 2008 Draft

© 2008 Richard L. Halterman. © Some rights reserved.

All the content of this document (including text, figures, and any other original works), unless otherwise noted, is licensed under a Creative Commons License.

Attribution-NonCommercial-NoDerivs

This document is free for non-commercial use with attribution and no derivatives.

You are free to copy, distribute, and display this text under the following conditions:

- **BY Attribution.** You must attribute the work in the manner specified by the author or licensor.
- **NC Noncommercial.** You may not use this work for commercial purposes.
- **ND No Derivative Works.** You may not alter, transform, or build upon this work.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

See <http://www.creativecommons.org> for more information.

Java is a registered trademark of Sun Microsystems, Incorporated.

DrJava Copyright © 2001-2003 was developed by the JavaPLT group at Rice University (javaplt@rice.edu).

Eclipse is an open-source project managed by the Eclipse Foundation (<http://www.eclipse.org>).

13 March 2008 Draft

Contents

1	The Context of Software Development	1
1.1	Software	1
1.2	Development Tools	2
1.3	The Java Development Environment	5
1.4	The <i>DrJava</i> Development Environment	9
1.5	Summary	11
1.6	Exercises	12
2	Values, Variables, and Types	14
2.1	Java Values in DrJava’s Interaction Pane	14
2.2	Variables and Assignment	17
2.3	Identifiers	22
2.4	Summary	24
2.5	Exercises	24
3	Arithmetic Expressions	26
3.1	Expressions and Mathematical Operators	26
3.2	String Concatenation	29
3.3	Arithmetic Abbreviations	30
3.4	Summary	31
3.5	Exercises	31
4	Objects: Packaging Computation	33
4.1	Classes	33
4.2	Comments	39
4.3	Local Variables	40
4.4	Method Parameters	42

4.5	Summary	43
4.6	Exercises	44
5	Boolean Expressions and Conditional Execution	46
5.1	Relational Operators	46
5.2	The <code>if</code> Statement	49
5.3	The <code>if/else</code> Statement	53
5.4	Summary	54
5.5	Exercises	55
6	More Complex Conditionals	56
6.1	Nested Conditionals	56
6.2	Multi-way <code>if/else</code> Statements	60
6.3	Errors in Conditional Expressions	64
6.4	Summary	64
6.5	Exercises	65
7	Modeling Real Objects	66
7.1	Objects with State	67
7.2	Traffic Light Example	70
7.3	<code>RationalNumber</code> Example	74
7.4	Object Aliasing	77
7.5	Summary	79
7.6	Exercises	80
8	Living with Java	82
8.1	Standard Output	82
8.2	Formatted Output	83
8.3	Keyboard Input	84
8.4	Source Code Formatting	85
8.5	Errors	87
8.6	Constants Revisited	91
8.7	Encapsulation	93
8.8	Summary	95
8.9	Exercises	96

9	Class Members	97
9.1	Class Variables	97
9.2	Class Methods	100
9.3	Updated Rational Number Class	100
9.4	An Example of Class Variables	104
9.5	The <code>main()</code> Method	105
9.6	Summary	106
9.7	Exercises	106
10	Composing Objects	109
10.1	Modeling Traffic Light Logic	109
10.2	Textual Visualization of a Traffic Light	113
10.3	Intersection Example	117
10.4	Summary	121
10.5	Exercises	122
11	Inheritance and Polymorphism	123
11.1	Inheritance	123
11.2	Protected Access	131
11.3	Visualizing Inheritance	132
11.4	Polymorphism	134
11.5	Extended Rational Number Class	135
11.6	Multiple Superclasses	137
11.7	Summary	137
11.8	Exercises	138
12	Simple Graphics Programming	140
12.1	2D Graphics Concepts	140
12.2	The <code>Viewport</code> Class	141
12.3	Event Model	146
12.4	Anonymous Inner Classes	148
12.5	A Popup Menu Class	150
12.6	Summary	151
12.7	Exercises	154
13	Some Standard Java Classes	155

13.1 Packages	156
13.2 Class System	156
13.3 Class String	158
13.4 Primitive Type Wrapper Classes	161
13.5 Class Math	163
13.6 Class Random	166
13.7 Summary	170
13.8 Exercises	170
14 The Object Class	171
14.1 Class Object	171
14.2 Object Methods	172
14.3 Summary	175
14.4 Exercises	176
15 Software Testing	177
15.1 The assert Statement	177
15.2 Unit Testing	178
15.3 JUnit Testing	179
15.4 Regression Testing	183
15.5 Summary	184
15.6 Exercises	185
16 Using Inheritance and Polymorphism	186
16.1 Abstract Classes	186
16.2 A List Data Structure	190
16.3 Interfaces	199
16.4 Summary	202
16.5 Exercises	202
17 Iteration	203
17.1 The while Statement	203
17.2 Nested Loops	207
17.3 Infinite Loops	208
17.4 Summary	212
17.5 Exercises	212

18 Examples using Iteration	213
18.1 Example 1: Drawing a Tree	213
18.2 Example 2: Prime Number Determination	215
18.3 Example 3: Digital Timer Display	219
18.4 A Mutable List	221
18.5 Summary	224
18.6 Exercises	224
19 Other Conditional and Iterative Statements	225
19.1 The <code>switch</code> Statement	225
19.2 The Conditional Operator	229
19.3 The <code>do/while</code> Statement	230
19.4 The <code>for</code> Statement	232
19.5 Summary	238
19.6 Exercises	238
20 Arrays	239
20.1 Declaring and Creating Arrays	241
20.2 Using Arrays	243
20.3 Arrays of Objects	248
20.4 Multidimensional Arrays	249
20.5 Summary	251
20.6 Exercises	252
21 Arrays as Objects	253
21.1 Array Parameters	253
21.2 Copying an Array	254
21.3 Array Return Values	258
21.4 Command Line Arguments	259
21.5 Variable Argument Lists	261
21.6 Summary	264
21.7 Exercises	264
22 Working with Arrays	265
22.1 Sorting Arrays	265
22.2 Searching Arrays	268

22.3 Summary	282
22.4 Exercises	282
23 A Simple Database	284
23.1 Summary	289
23.2 Exercises	289
24 Graphical Objects	290
24.1 The <code>GraphicalObject</code> Class	290
24.2 Graphical Text Objects	295
24.3 Graphical Buttons	296
24.4 Summary	296
24.5 Exercises	296
25 Exceptions	297
25.1 Exception Example	298
25.2 Checked and Unchecked Exceptions	305
25.3 Using Exceptions	307
25.4 Summary	309
25.5 Exercises	309
Bibliography	310
Index	311

List of Tables

2.1	Primitive data types	17
2.2	Structure of declaration statements	21
2.3	Java reserved words	23
3.1	Arithmetic operators	26
5.1	Relational operators	46
5.2	Logical operators	48
8.1	Control codes used in the <code>System.out.printf()</code> format string	84
8.2	A subset of the methods provided by the <code>Scanner</code> class	85
12.1	A subset of the drawing methods provided by the <code>Viewport</code> class, a rectangular drawing surface. These methods should not be used outside of the <code>draw()</code> method.	143
12.2	Viewport events and their corresponding methods	146
13.1	A subset of the methods provided by the <code>String</code> class	159
13.2	Wrapper classes used to objectify primitive types	162
13.3	Some useful <code>Integer</code> methods	162
13.4	Some useful <code>Math</code> methods	164
13.5	Some useful <code>Random</code> methods	167
14.1	Two commonly used <code>Object</code> methods	172
20.1	Zero-like values for various types	242
21.1	Empirical data comparing a straightforward array copy to <code>System.arraycopy()</code>	257
22.1	Empirical data comparing selection sort to Quicksort	269
22.2	Empirical data comparing linear search to binary search	276

22.3 A structural analysis of the linear search method	278
22.4 A structural analysis of the binary search method	280

List of Figures

1.1	Program development cycle for typical higher-level languages	3
1.2	Java technology architecture	7
1.3	Java program development cycle	8
1.4	DrJava	10
1.5	Eclipse	12
4.1	Formulas for the circumference and area of a circle given its radius	33
4.2	Formulas for the perimeter and area of a rectangle given it length and width.	38
5.1	Execution flow in an <code>if</code> statement	50
5.2	Execution flow in an <code>if/else</code> statement	54
7.1	A simple traffic light to be modeled in software	71
7.2	Assignment of Primitive Types	79
7.3	Assignment of Reference Types	80
8.1	Result of compiling a Java source file with a compile-time error	88
10.1	Component relationships within traffic light model objects	113
11.1	UML diagram for the composition relationships involved in the <code>Intersection</code> class	133
11.2	UML diagram for the traffic light inheritance relationship	133
11.3	UML diagram for the test intersection dependencies	133
11.4	UML diagram for classes used in <code>TestIntersection</code>	134
12.1	The graphics coordinate system	141
12.2	Graphical traffic light design	144
13.1	String indexing	160

13.2	String reassignment	161
13.3	A problem that can be solved using methods in the <code>Math</code> class.	164
13.4	A pair of dice	167
15.1	Results of a passed JUnit test	181
15.2	Results of a failed JUnit test	182
16.1	Class hierarchy of shape classes	189
16.2	Class hierarchy of fruit classes	190
16.3	Class hierarchy of list classes	191
16.4	Data structure created by the interactive sequence	197
17.1	Execution flow in a <code>while</code> statement	205
19.1	Execution flow in a <code>do/while</code> statement	231
19.2	Execution flow in a <code>for</code> statement	234
20.1	Creating arrays	243
20.2	Assigning an element in an array	244
20.3	A 2×3 two-dimensional array	250
20.4	A “jagged table” where rows have unequal lengths	251
21.1	Array aliasing	255
21.2	Making a true copy of an array	256
21.3	Correspondence of command line arguments to <code>args</code> array elements	261
22.1	Linear search of an array	272
22.2	Binary search of an array	273
22.3	Comparison of linear search versus binary search	277
22.4	Plot of the mathematical functions obtained through the code analysis	283
25.1	The hierarchy of <code>Exception</code> classes	306

List of Classes

1.1	RemainderTest—Prints the remainder of $15 \div 6$	10
4.1	CircleCalculator—a class used to create CircleCalculator objects that can compute the circumference and area of circles	34
4.2	RectangleCalculator—a class used to create RectangleCalculator objects that can compute the areas and perimeters of rectangles	38
4.3	TimeConverter—converts seconds to hours, minutes, and seconds	41
5.1	TimeConverter2—converts seconds to hours:minutes:seconds	52
6.1	CheckRange—Checks to see if a value is in the range 0–10	56
6.2	NewCheckRange—Checks to see if a value is in the range 0–10	57
6.3	EnhancedRangeCheck—Accept only values within a specified range and provides more comprehensive messages	57
6.4	BinaryConversion—Builds the 10-bit binary equivalent of an integer supplied by the user	59
6.5	SimplerBinaryConversion—Re-implements BinaryConversion with only one <i>if</i> statement	60
6.6	DigitToWord—Multi-way <i>if</i> formatted to emphasize its logic	61
6.7	RestyledDigitToWord—Reformatted multi-way <i>if</i>	62
6.8	DateTransformer—Transforms a numeric date into an expanded form	63
7.1	Circle1—a slightly different circle class	67
7.2	Circle2—adds a new constructor to the Circle1 class	69
7.3	Circle3—one constructor calls another	70
7.4	RYGTrafficLight—simulates a simple traffic light	73
7.5	RationalNumber—defines a rational number type	75
7.6	SimpleObject—a class representing very simple objects	77
8.1	SimpleInput—adds two integers	85
8.2	ReformattedCircleCalculator—A less than desirable formatting of source code	86
8.3	CircleCalculatorWithMissingBrace—A missing curly brace	88
8.4	MissingDeclaration—A missing declaration	89
8.5	PotentiallyDangerousDivision—Potentially dangerous division	90
8.6	BetterDivision—works around the division by zero problem	91
9.1	Rational—updated version of the RationalNumber class	102
9.2	Widget—Create widgets with unique, sequential serial numbers	104
9.3	VerySimpleJavaProgram—a very simple Java program	106
10.1	TrafficLightModel—final version of the traffic light class	110
10.2	TextTrafficLight—a text-based traffic light visualization	114
10.3	VerticalTextLight—an alternate text-based traffic light visualization	116
10.4	Intersection—simulates a four-way intersection of simple traffic lights	119
10.5	IntersectionMain—simulates a four-way intersection of simple traffic lights	121
11.1	TurnLightModel—extends the TrafficLightModel class to make a traffic light with a left turn arrow	124
11.2	TextTurnLight—extends the TextTrafficLight class to make a traffic light with a left turn arrow	126
11.3	TestIntersection—provides some convenience methods for creating two kinds of intersections	129

11.4	EnhancedRational—extended version of the Rational class	137
12.1	DrawTrafficLight—Renders a graphical traffic light	145
12.2	DrawStar—draws a five-pointed star	146
12.3	SimpleResponder—monitors two kinds of events—mouse button presses and keystrokes	147
12.4	InteractiveTrafficLightViewport—a simple interactive graphical traffic light	148
12.5	UsingSimpleResponder—uses the SimpleResponder class.	149
12.6	AnonymousSimpleResponder—avoids the use of the SimpleResponder class.	149
12.7	SimpleMenuExample—uses a popup menu to modify the displayed number.	151
13.1	Stopwatch—implements a software stopwatch to time code execution	157
13.2	StringExample—Illustrates some string methods	159
13.3	IntegerTest—Exercises the Integer class	163
13.4	OrbitDistanceTable—Generates a partial table of the orbital position and distance data	166
13.5	DieGame—simulates rolling a pair of dice	168
13.6	Die—simulates a die	169
14.1	EqualsTest—illustrates the difference between equals() and ==	173
14.2	InstanceOfTest—demonstrates the use of the instanceof operator	175
15.1	TrafficLightCustomTester—custom class for testing traffic light objects	179
15.2	TrafficLightTester—testing traffic light objects	180
15.3	BetterTrafficLightTester—more focused unit tests	183
16.1	Circle1a—a slightly modified Circle1	186
16.2	Rectangle1a—a stateful rectangle	187
16.3	PerimeterToAreaRatio—computes the perimeter-to-area ratio of a geometric shape	187
16.4	Shape—a generic shape	188
16.5	Circle—a circle subclass	189
16.6	Rectangle—a rectangle subclass	189
16.7	List—Abstract superclass for list objects	192
16.8	EmptyList—Class for empty list objects	193
16.9	NonemptyList—Class for nonempty list objects	195
16.10	ListTest—JUnit test file	199
16.11	IShape—an interface for shape objects	200
16.12	Ring—a ring (circle) class implementing the shape interface	200
16.13	Quadrilateral—a quadrilateral (rectangle) class implementing the shape interface	200
16.14	ShapeReport—Illustrates the <i>is a</i> relationship for interfaces	201
17.1	CountToFive—Simple count to five	203
17.2	IterativeCountToFive—Better count to five	204
17.3	AddUpNonnegatives—Sums any number of nonnegative integers	206
17.4	AddUpNonnegativesSimpler—Sums any number of nonnegative integers	206
17.5	TimesTable—Prints a multiplication table	207
17.6	FindFactors—an erroneous factoring program	209
18.1	StarTree—Draw a tree of asterisks given a user supplied height	214
18.2	PrintPrimes—Prime number generator	215
18.3	InstrumentedPrimes—Instrumented prime number generator	218
18.4	DigitalTimer—a digital timer program that accurately keeps time	220
18.5	ListNode—a primitive building block for mutable lists	222
18.6	MutableList—a class for building mutable list objects	223
18.7	MutableListTest—testing our mutable list	224
19.1	SwitchDigitToWord—switch version of multi-way if	228
19.2	GoodInputOnly—Insist the user enter a good value	230
19.3	BetterInputOnly—GoodInputOnly (19.2) using a do/while loop	232
19.4	IntRange—a useful reusable input method	232

19.5	ForCounter—IterativeCountToFive (17.2) using a for statement in place of the while . . .	233
19.6	BetterTimesTable—Prints a multiplication table using for statements	234
19.7	ForPrintPrimes—PrintPrimes using for loops	235
20.1	AverageNumbers—Average five numbers entered by the user	239
20.2	AverageNumbers2—Another program that averages five numbers entered by the user	240
20.3	ArrayAverage—Use an array to average five numbers entered by the user	245
20.4	BetterArrayAverage—An improved ArrayAverage	246
21.1	ArrayCopyBenchmark—compares a straightforward array copy to System.arraycopy()	257
21.2	PrimesList—Uses a method to generate a list (array) of prime numbers over a given range	259
21.3	CmdLineAverage—Averages values provided on the command line	260
21.4	OverloadedSumOfInts—A simple class to sum integers	262
21.5	ArraySumOfInts—A “better” class to sum integers using an array	262
21.6	VarargsSumOfInts—A truly better class to sum integers using varargs	263
21.7	VarargTester—A truly better class to sum integers using varargs	263
22.1	SortIntegers—Sorts the list of integers entered by the user on the command line	266
22.2	CompareSorts—Compares the execution times of selection sort versus Quicksort	268
22.3	LinearSearch—finds an element within an array	270
22.4	SearchCompare—Compares the performance of linear search versus binary search	275
22.5	Log2—demonstrates that $\log_2 x$ corresponds to determining how many times x can be divided in half	281
23.1	EmployeeRecord—information about a single employee	285
23.2	Database—uses an array to store a collection of employee records	287
23.3	EmployeeDatabase—allows a user to interactively exercise the employee database	289
24.1	GraphicalStarObject—a simple star-shaped graphical object	291
24.2	SimpleStar—uses the star object	292
24.3	MovableStar—allows the user to drag the star around in the viewport	292
24.4	MovableCompositeStar—graphical objects can be composed of other graphical objects	293
24.5	ChangableStar—graphical objects with popup menus	294
24.6	StarMenu—a context-sensitive popup menu for modifying GraphicalStarObjects	295
24.7	TextGraphics—uses some movable and immobile graphical text objects	296
25.1	Collection.InvalidCollectionSizeException—Thrown when creating a Collection with a nonpositive size	299
25.2	Collection.CollectionFullException—Thrown when attempting to insert an element into a full Collection	299
25.3	Collection.Collection—A simple data structure that uses exceptions	300
25.4	CollectTest—Exercises the Collection class	303

Preface

The purpose of this text is to introduce computer programming to a diverse audience. It uses Java as the programming medium since the language provides a relatively safe environment for beginning students to learn programming concepts while offering a rich platform capable of building industrial-strength software systems. Students therefore can be exposed quickly to object-oriented principles with the assurance that, if necessary, they later can build upon these experiences and do more serious software development without having to learn another programming language.

Concerns exist about using Java as the first programming language. Many have observed that beginners must use some sophisticated language features before they understand what they mean just to write simple programs. The Association for Computing Machinery's Special Interest Group on Computer Science Education (ACM-SIGSCE) commissioned the ACM Java Task Force to explore and address these concerns. It proposed a new style of Java programming using a custom class library.

This text tries to strike a balance between standard Java coding idioms and simplified

This part is incomplete.

Chapter 1

The Context of Software Development

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer's memory and interact with the peripheral devices like the display screen in such a way as to produce the “magic” that permits humans to perform useful tasks and solve high-level problems. One program allows a personal computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

- at the lower, more concrete level electrical impulses alter the internal state of the computer, while
- at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract, level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Sophisticated tools hide the lower-level details from the programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as *software engineers* develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers. This chapter examines the context of software development, including computer systems and tools.

1.1 Software

A computer program is an example of computer *software*. One can refer to a program as a *piece* of software as if it were a tangible object, but software is actually quite intangible. It is stored on a *medium*. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used software must be stored in the computer's memory. Typically computer programs are loaded into memory from a medium like the computer's hard disk. The electromagnetic

pattern stored on the hard drive is transferred to the computer's memory where the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system:

...10001011011000010001000001001110...

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that a particular point on the screen will be colored red. Unfortunately, only a miniscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running Microsoft Windows. Further, almost none of those who could produce the binary sequence would claim to enjoy the task. The Word program for Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Pentium processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own *machine language*.

1.2 Development Tools

If very few humans can (or want) to speak the machine language of the computers' processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that for humans are easier to manage than binary sequences. Tools exist that convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Java, C++, C#, FORTRAN, COBOL, and Perl allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Most programmers today, especially those concerned with high-level applications, do not usually worry about the details of underlying hardware platform and its machine language.

One might think that ideally a tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called *compilers* that readily translate one computer language into another have been around for over 50 years, but natural language recognition is still an active area of artificial intelligence research. Natural languages, as they are used by most humans, are inherently ambiguous. To properly understand all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today's software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any program that can be solved by a computer.

Consider the following program fragment, written in the C programming language:

```
subtotal = 25;
tax = 3;
total = subtotal + tax;
```

(The Java language was derived from C.) The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, *subtotal*, *tax*, and *total*, called *variables*, are used

to hold information (§ 2.2).¹ Variables have been used in mathematics for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer's memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the C language, not machine language, it cannot be executed directly on any processor. A program called a *compiler* translates the C code to machine code. The typical development cycle for higher-level programming languages is shown in Figure 1.1.

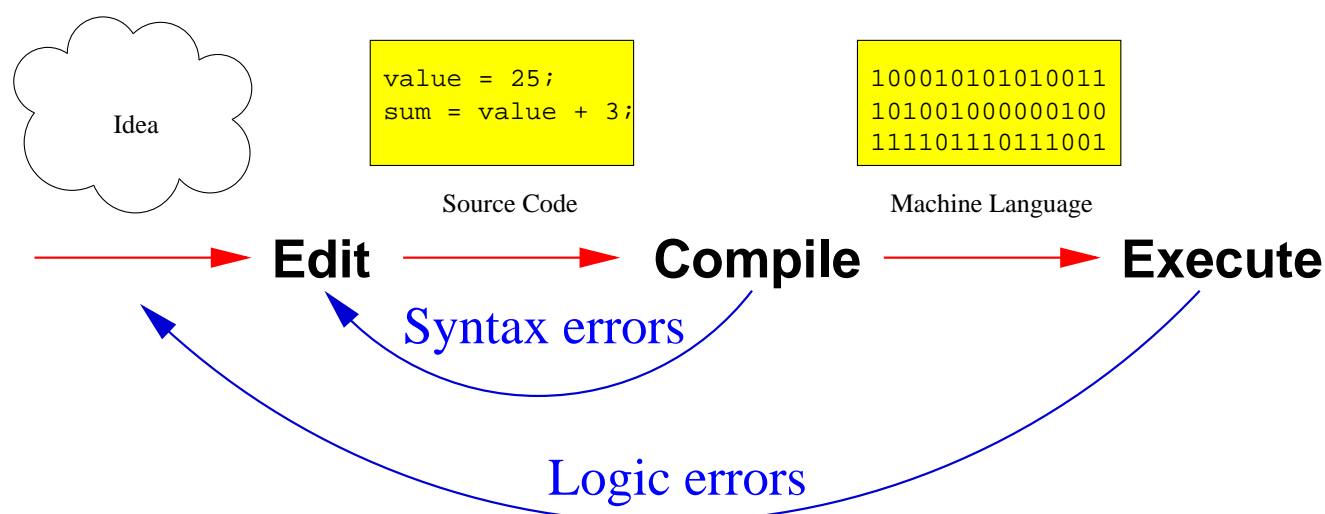


Figure 1.1: Program development cycle for typical higher-level languages. Errors are discussed in Section 8.5.

The higher-level language code is called *source code*. The compiled machine language code is called the *target code*.

The beauty of higher-level languages is that the same C source code can be compiled to different target platforms and execute identically. The only requirement is that the target platform has a C compiler available. So, a program can be developed on one system and then recompiled for another system. The compiled code will be very different on the various platforms since the machine architecture varies. Importantly, the human programmer thinks about writing the solution to a problem in C, not in a specific machine language.

For true compatibility, the program to be transported to multiple platforms should not exploit any feature unique to any target platform; for example, graphical windows behave differently and have different capabilities under different window managers (for example, the Windows XP graphical environment is different from the Unix/Linux X windows environment and Mac OS X environment). In practice, programs are often decomposed into platform *dependent* parts and platform *independent* parts. The platform independent parts capture the core functionality or purpose of the software (sometimes called the *business logic*). For example, all spreadsheets have some things in common (cells, formatting, calculation, etc.), regardless of the platform upon which they run. The platform dependent parts of the program take care of the unique user interface or file structure of the target platform. Porting the application to another platform requires modifying only the platform dependent parts of the software.

The ability to develop a program once and deploy it easily on other platforms is very attractive to developers since software development is an expensive and time-consuming task. § 1.3 shows how Java provides a new dimension to software portability.

As mentioned above, computer programs can be written in the absence of actual computers on which to run them. They can be expressed in abstract mathematical notation and be subjected to mathematical proofs and analysis that verify their correctness and execution efficiency. Some advantages of this mathematical approach include:

¹The section symbol (§) is used throughout the text to direct you to sections that provide more information about the current topic. The listing symbol (▢) refers to a source code listing such as a Java class definition.

- **Platform independence.** Since the program is not expressed in a “real” programming language for execution on a “real” computer system, the programmer is not constrained by limitations imposed by the OS or idiosyncrasies of the programming language.
- **Avoids the limitations of testing.** Testing cannot prove the absence of errors; it can only demonstrate the presence of errors. Mathematical proofs, when done correctly, are not misled by testing traps such as:
 - Coincidental correctness—as a simple example, consider the expression $x + y$. If the programmer accidentally uses multiplication instead of addition and values $x = 2$ and $y = 2$ are used to check, then the program appears to evaluate this expression correctly, since both $2 + 2 = 4$ and $2 \times 2 = 4$.
 - Poor coverage—in complex programs, certain parts of a program may be executed only under unusual conditions. Often these unusual conditions elude testing, and lurking errors appear only after the software is deployed.

Unlike pure mathematicians, most programmers do not often use formal mathematical reasoning techniques as they write programs. Some argue, with a strong case, that this lack of formality leads to poor-quality software. However, capable programmers employ techniques that are often far less formal but highly effective in producing high-quality software. These informal techniques and notions often map directly to their more abstract mathematical counterparts. Some circumstances, such as the development of critical software systems concerning safety, privacy, or financial affairs, may warrant a more formal approach.

Programmers have a variety of concrete tools available to enhance the software development process. Some common tools include:

- **Editors.** An *editor* allows the user to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.
- **Compilers.** A *compiler* translates the source code to target code. The target code may be machine language for a particular platform, another source language, or, as in the case of Java, instructions for a *virtual machine* (§ 1.3).
- **Debuggers.** A *debugger* allows programmers to simultaneously run a program and see which source code line is currently being executed. The values of variables and other program elements can be watched to see if their values change as expected. Debuggers are valuable for locating errors (also called *bugs*) and repairing programs that contain errors. (See § 8.5 for more information about programming errors.)
- **Profilers.** A *profiler* is used to evaluate a program’s performance. It indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also are used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. (This is known as *coverage*. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing.)

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of IDEs include *Eclipse*, Microsoft’s *Visual Studio*, *NetBeans*, and *DrJava*.

Despite the plethora of tools (and tool vendors’ claims), the programming process for all but trivial programs is not automatic. Good tools are valuable and certainly increase the productivity of developers, but they cannot write software. There are no substitutes for sound logical thinking, creativity, common sense, and, of course, programming experience.

1.3 The Java Development Environment

Java was conceived in the early 1990s to address some outstanding issues in software development:

- **Computing platforms were becoming more diverse.** Microprocessors were becoming ubiquitous for controlling electronic devices like televisions, cell phones, and home bread makers. These computers, called *embedded computers*, were driven, of course, by software. Meeting the software needs of this multitude of embedded computers had become quite a task. A uniform platform for software development is desirable from one point of view, but forcing all manufacturers to use the same hardware is impractical.
- **Embedded computers had to be robust.** Consumers should not have to reboot their television or toaster because its control software has crashed. These appliances must be as reliable as their noncomputerized predecessors.
- **Computing platforms were becoming more distributed.** In order to accomplish a task, a system may need to download software from a remote system. For example, the functionality of a web browser can be extended by downloading special software called a *plug-in*. Cell phone service may be enhanced by upgrading some or all of its existing software. In general, though, downloading arbitrary software from a remote machine is risky. The software may do damage to the local machine either intentionally (like a virus) or accidentally (due to an error in the software). A framework was needed that would provide a safer environment in which untrusted code could run.
- **Object-oriented techniques had revolutionized software development.** Object-oriented (OO) programming languages and development environments, in the hands of knowledgeable developers, facilitate the construction of complex and large software systems. OO development had been embraced by many applications developers, but non-OO languages like C and assembly language still dominated systems programming.

No existing development environment at the time adequately addressed all these issues. Engineers at Sun Microsystems developed Java to address these issues. Around the same time as Java's inception the World Wide Web was emerging. Like the software appliances described above, Web software needed to be distributed, robust, and secure. The design of the Java programming language matched well with the needs of Web software developers.

Roughly speaking, web-based software can be classified into two types:

- **Client side software.** This software executes on the *client's* computer. A web browser is an example of a client; usually many clients can access a single website.
- **Server side software.** This software executes on the *server* computer. The computer providing the web page that clients (browsers) read is the server. Usually one server sends information to many clients.

Java was designed to address the issues of distributed computing, including security. Ordinarily, a program is loaded and executed by the operating system (OS) and hardware. Java programs, however, do not execute *directly* on the client OS and hardware. Java code is executed in a special environment called the *Java Runtime Environment* (JRE). The JRE provides several features that ensure secure computing on the target platform:

- **The Java Virtual Machine (JVM).** The JVM is an executing program that emulates a computer that understands a special machine language called Java *bytecode*. Compiled Java programs run on the JVM, not directly on the native processor of the computer, so the JVM can be designed to prohibit any insecure behavior of executing programs.
- **Run time error detection.** The JRE detects many types of errors that an executing program may exhibit. For example, an arithmetic calculation may attempt to divide a number by zero.

- **Memory management.** A memory manager controls memory access, allocation, and deallocation. The memory manager ensures only memory assigned to the JVM is affected.
- **Security.** A security manager enforces policies that limit the capabilities of certain Java programs. The spectrum ranges from untrusted applets downloaded from the web that are not allowed to do any damage to a client's system to Java applications that have the same abilities as any other software.

The JRE includes hundreds of predefined Java components that programmers can incorporate into their programs. These components consist of compiled Java bytecode contained in units called *classes*. These classes, called the standard library classes, provide basic functionality including file manipulation, graphics, networking, mathematics, sound, and database access.

The virtual machine concept is so attractive for some applications that compilers are available that translate other higher-level programming languages such as Python, Lisp, and SmallTalk into bytecode that executes on the Java Virtual Machine.

The JRE is itself software. Reconsider the power of software. Software can turn a computer into a flight simulator; that is, a computer can emulate a plane in flight. Simulated controls allow a virtual pilot (the user) to adjust the virtual altitude, airspeed, and attitude. Visual feedback from instruments and a simulated viewport mimic the plane's response to the pilot's input. The laws of physics in this virtual world so closely match those in the real world that the pilot is given the illusion of real flight. If software can so closely model this complex physical activity as well as it does, it should not be surprising that software can cause the same computer to behave as if it were a different kind of computer; that is, it can execute machine language instructions meant for an entirely different processor. This process of one computer behaving as if it were another is called *emulation*. This concept of emulation is not new, but it is central to the Java execution environment. Java's approach to software development and deployment is slightly different from the standard approach:

- **Traditional approach**

1. Develop the program in a higher-level language.
2. For each deployment platform, recompile the source code using a compiler for that target platform.

- **Java's approach**

1. Develop the program in a higher-level language (Java).
2. Compile the source code to a special machine language called *bytecode*.
3. For each deployment platform, execute the compiled bytecode using a special emulator built for that target platform. This emulator is called the Java Virtual Machine, or JVM.

Figure 1.2 illustrates the Java technology architecture.

As Figure 1.2 illustrates, the Java development environment works as follows:

1. Applications and applet programmers develop Java source code and store this code in files.
2. The Java compiler translates these source files into bytecode class files.
3. The JVM:
 - loads and executes the bytecode instructions in these programmer-developed class files as needed by the running program,
 - loads any standard library classes required by the program, and

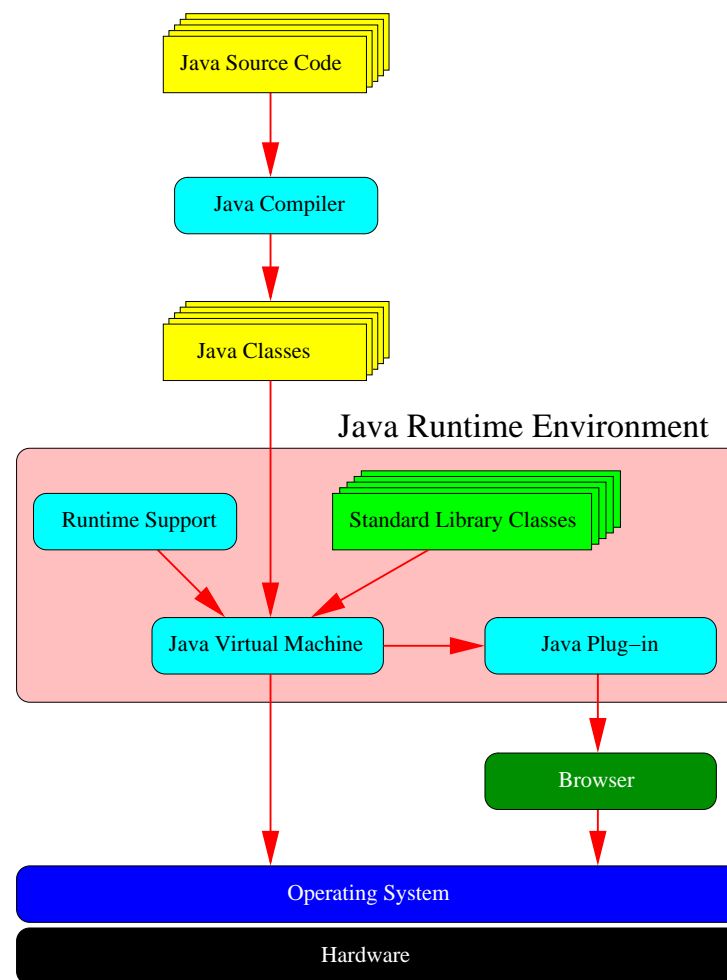


Figure 1.2: Java technology architecture.

- executes runtime support code that handles assorted tasks like memory management, thread scheduling and run time error checking.
4. The JVM interprets the bytecodes and interacts with the OS using the native machine language of the platform so that the effects of the executing Java program can appear on the user's computer system.
 5. The OS, as dictated by the JVM, directs the hardware through device drivers. For example,
 - input received from the keyboard or mouse is directed to the JVM
 - the graphics requested by the JVM appear on the screen
 - files created by the Java program are written to a disk drive.

A *Java plug-in* provides the same JRE to a browser so Java programs called *applets* can be downloaded from the Web and safely executed within the context of the browser. For most web users this process is transparent—most users will visit a web page that displays some interesting effects and be unaware that a Java program has been downloaded and began executing within their browser window.

The Java approach has several advantages:

- **Write once, run anywhere.** A program can be written once, compiled once, and run on any platform for which a Java Virtual Machine exists. In the traditional approach (outlined in § 1.2) this degree of portability is rare.

- **Security.** The implementation of the JVM can provide security features that make it safer to execute code downloaded from remote machines. For example, applets downloaded from a network are subject to security checks and are not allowed to read from or write to clients' disk drives. The JVM includes a security manager that enforces such policies.

One disadvantage of the virtual machine approach is that a Java bytecode program generally will run slower than an equivalent program compiled for the target platform. This is because the bytecode is not executed directly by the target platform. It is the JVM that is executing directly on the target platform; the JVM executes the bytecode. The JVM, therefore, adds some overhead to program execution. The HotSpot™ performance engine JVM addresses this issue by optimizing the bytecode as the program executes; parts of the code that execute frequently are translated into native machine language and executed directly by the processor. Other optimizations are also performed. The HotSpot JVM has significantly reduced the performance disadvantage in recent years.

If a JVM is unavailable for a particular platform, the Java bytecode cannot be executed on that platform. Equivalently, from the traditional approach, if a compiler for a given language is unavailable for a particular platform, the code cannot be ported directly to and executed on that platform.

While Java's special appeal is web-based programming, Java can be used for developing traditional applications as well. Like applets, Java applications typically are executed by the JVM. The Java program development cycle, shown in Figure 1.3, is similar to the development cycle for traditional compiled languages, as shown in Figure 1.1.

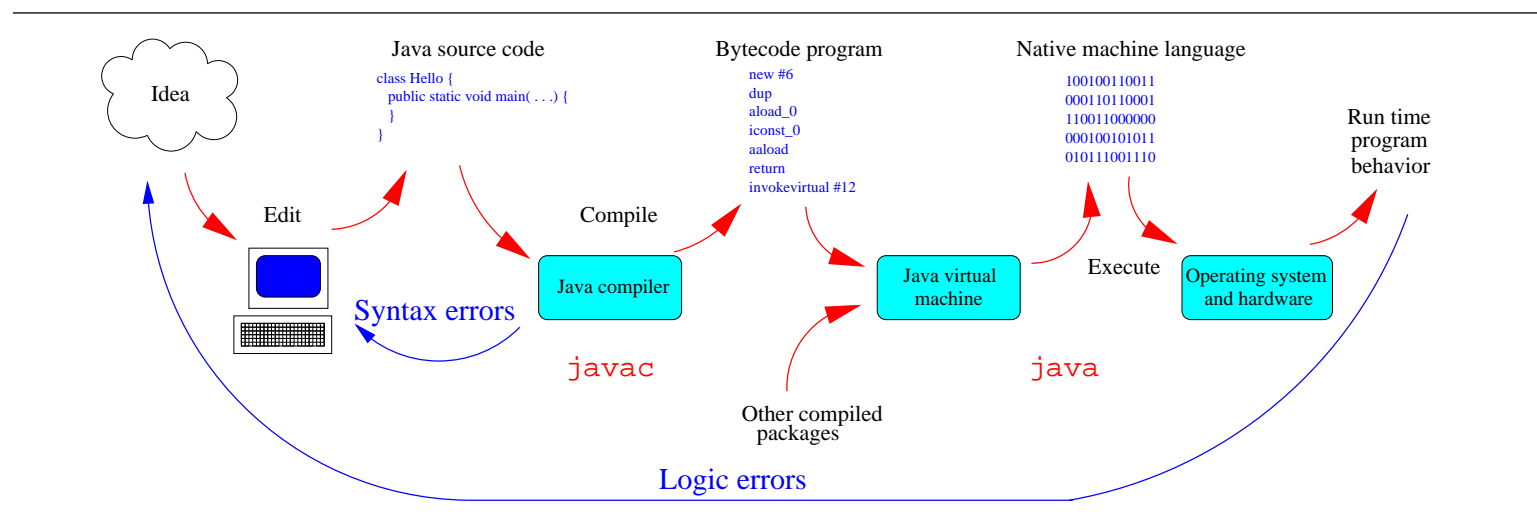


Figure 1.3: Java program development cycle

Sun Microsystems provides a set of tools for Java development, called the Java Software Developers Kit (Java SDK). These tools are freely available for noncommercial use from <http://java.sun.com>. The SDK includes a JRE, but the JRE is available separately for nondevelopers wishing only to run Java programs. Microsoft Windows, Linux, and Solaris platforms are supported. The Apple Mac OS X platform has the Java tools available from Apple. The code in this book is based on features available in the Java 5.0 (also called Java 2 SDK 1.5.0) SDK.

Sun Microsystems owns the intellectual property rights to Java and Java-related technology and defines strict standards for its implementation. Vendors' Java tools must pass stringent compatibility tests before they may be distributed. As examples:

- A JRE implementation must provide the complete set of standard library classes. If this were not guaranteed, then a Java program expecting to use a standard class would not be able to execute.
- A Java compiler must
 1. accept any correct Java program and translate it into bytecode executable on a JVM

2. reject all incorrect Java programs

This prevents vendors from adding features to the language producing an incompatible version of Java.

Such standards ensure that Java programs truly can be “write once, run anywhere.”

For more on the history of Java see

- <http://www.java.com/en/about/>
- <http://www.java.com/en/javahistory/>

For more information about Java technology see

<http://java.sun.com/docs/books/tutorial/getStarted/intro/index.html>.

1.4 The *DrJava* Development Environment

Unlike some programming languages used in education, Java is widely used in industry for commercial software development. It is an industrial strength programming language used for developing complex systems in business, science, and engineering. Java drives the games on cell phones and is used to program smart cards. Java is used to manage database transactions on mainframe computers and is used to enhance the user experience on the Major League Baseball’s website. NASA uses Java in its command and control software for the Mars Rover. Java is used for financial analysis in Wall Street investment firms.

In order to accomplish all that it does, Java itself is complex. While experienced programmers can accomplish great things with Java, beginners sometimes have a difficult time with it. Programmers must be aware of a number of language features unrelated to the task at hand in order to write a relatively simple program. These features intimidate novices because they do not have the background to understand what the features mean and why the features are required for such simple programs. Professional software developers enjoy the flexible design options that Java permits. For beginners, however, more structure and fewer options are better so they can master simpler concepts before moving on to more complex ones.

Despite the fact that novice programmers struggle with its complexity, Java is widely used in first-year programming courses in high schools, colleges, and universities. Many instructors believe it to be a cleaner, safer language for beginners than other popular options such as C++, another object-oriented language widely used in industry.

DrJava (<http://drjava.sourceforge.net>) is a Java development environment created especially for students. Its development by the faculty and students at Rice University is ongoing. *DrJava* presents a simple user interface containing a multipane editor, a simple file manager, an interactive area, console display window, and compiler message area. It also has a built in debugger. *DrJava* lacks many of the amenities provided by other Java IDEs, but these other popular IDEs provide a vast array of features that overwhelm beginning programmers. While an introductory programming student may be able to use only 10% of the features provided by an industrial strength IDE, most of *DrJava* can be mastered quickly. Figure 1.4 is a screenshot of *DrJava*.

It is important to realize the difference between Java and *DrJava*. Java is a programming language and associated runtime environment; *DrJava* is a tool for exploring Java’s capabilities and developing Java software. *DrJava* itself was written in Java.

Java is used widely for commercial software development. In order to support this wide variety of applications development, the Java language is inherently complex. In fact, one cannot write the simplest Java program possible without using several advanced language constructs. For example, suppose we want to experiment with the Java

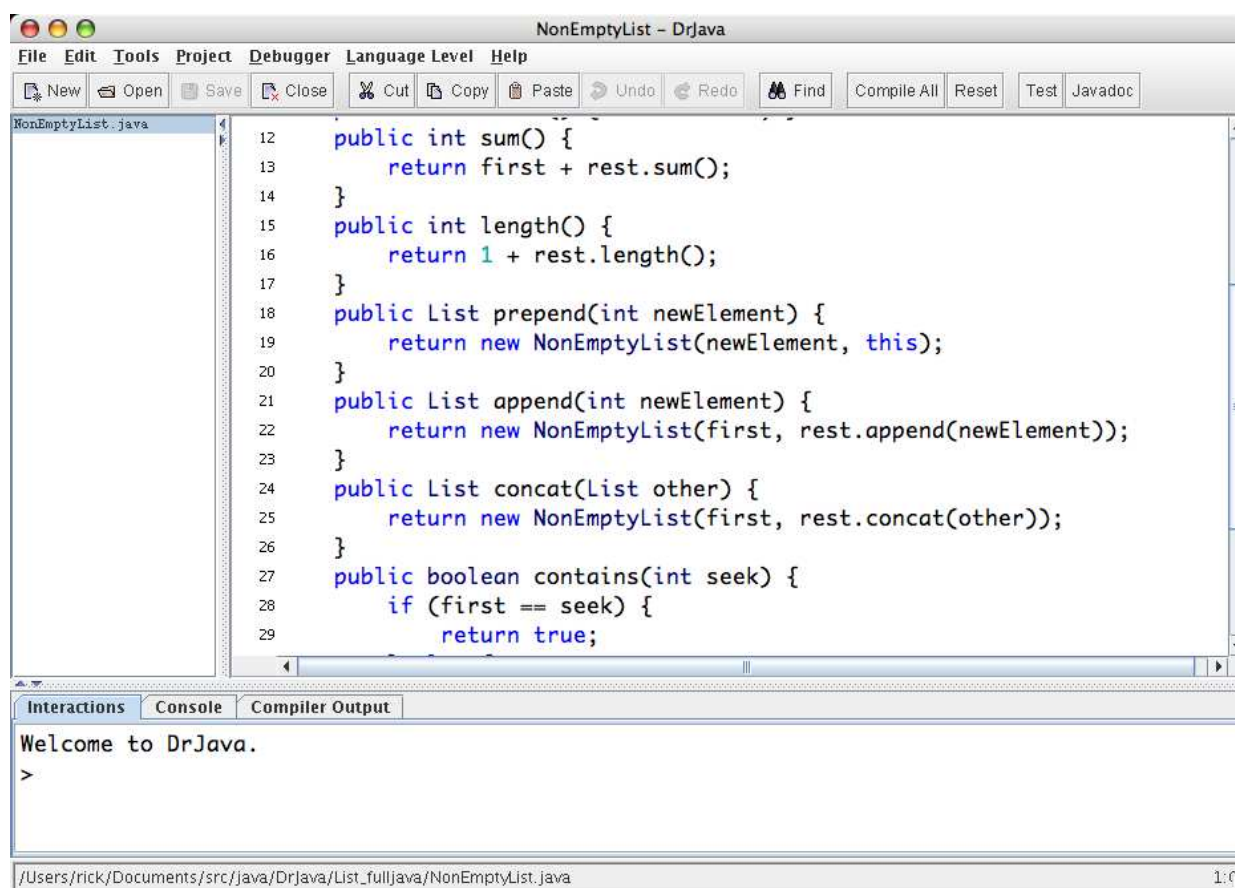


Figure 1.4: DrJava. Shown here is a screenshot of DrJava.

modulus (%) operator. Applied to integer values, the modulus operator computes the remainder of a division operation. RemainderTest (Figure 1.1) is one of the simplest Java programs that computes and displays the remainder of $15 \div 6$.

```
public class RemainderTest {
    public static void main(String[] args) {
        System.out.println(15 % 6); // Compute remainder
    }
}
```

Listing 1.1: RemainderTest—Prints the remainder of $15 \div 6$

We must type RemainderTest (Figure 1.1) within an editor, compile it, and then execute it and examine the results. This is a lot of work just to satisfy our curiosity about a simple calculation. Besides, what does the word `static` mean? Why is `void` there? These words are required for a complete, legal Java program, but considerable programming experience is necessary to fully understand and appreciate their presence.

To try out the operation on a different pair of values (for example, what if one or both of the numbers are negative?) we must edit the file, recompile it, and rerun it.

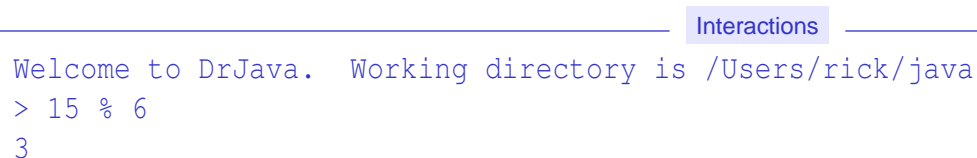
For beginning programmers learning the Java language this need to use advanced constructs for even the simplest of programs presents a problem with only two less-than-perfect solutions:

1. attempt to explain briefly these advanced features in terms that a novice could not understand and fully appreciate, or
2. announce that you do it this way “just because,” and the secrets will be revealed later as more knowledge and experience is gained.

Neither of these solutions is very satisfying. The first can unnecessarily confuse students with too much, too soon. The second can lead students to believe that they really cannot understand fully what is going on and, therefore, some kind of knowledge known only to gurus is needed to program computers.

DrJava solves this dilemma by providing an interactive environment where simpler concepts can be examined without the clutter of more advanced language features required by standalone programs. This actually allows us to investigate some more advanced features of Java earlier than we otherwise might since we can avoid becoming bogged down in the required (for standalone programs) but distracting details.

How could you see the results of Java’s interpretation of $15 \div 6$ under DrJava? Within DrJava’s Interactions pane type:



```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> 15 % 6
3
```

(The indicated working directory will be specific to your computer’s environment.) The Interactions pane prints `>`, and you type `15 % 6`. The DrJava interpreter then computes and displays the result, `3`.

Running in the Interactions pane is a Java interpreter that evaluates Java statements as you type them. This interactive environment is an ideal place to learn the basics of the Java language that we will later use to build more powerful Java programs. All the distracting details required to write “real” Java programs will be revealed eventually when the time comes.

Some students may find DrJava’s IDE to be a bit too simplistic. The Eclipse IDE (<http://www.eclipse.org>) is used by professional software developers all over the world. It provides a rich set of features that increase the productivity of experienced developers. Some examples include code completion, refactoring tools, advanced code analysis, and the ability to correct simple coding mistakes. Furthermore, Eclipse is designed to allow tool developers to build additional features that can be plugged into its framework.

The DrJava team has constructed a DrJava plug in for Eclipse. This allows students to use a more powerful IDE and still enjoy the interactive environment provided by DrJava. Figure 1.5 is a screenshot of Eclipse. Timid students may find Eclipse to be overwhelming and wish to stick with DrJava’s IDE.

Our explorations in this book, especially the earlier sections, assume you have access to the latest versions of DrJava (or Eclipse with the DrJava plug in) and Sun’s Java 5.0 development tools (also known as JDK 1.5.0).

1.5 Summary

- Computers require both hardware and software to operate. Software consists of instructions that control the hardware.
- Application software can be written largely without regard to the underlying hardware. A tool called a compiler translates the higher-level, abstract language into the machine language required by the hardware.
- Programmers develop software using tools such as editors, compilers, debuggers, and profilers.

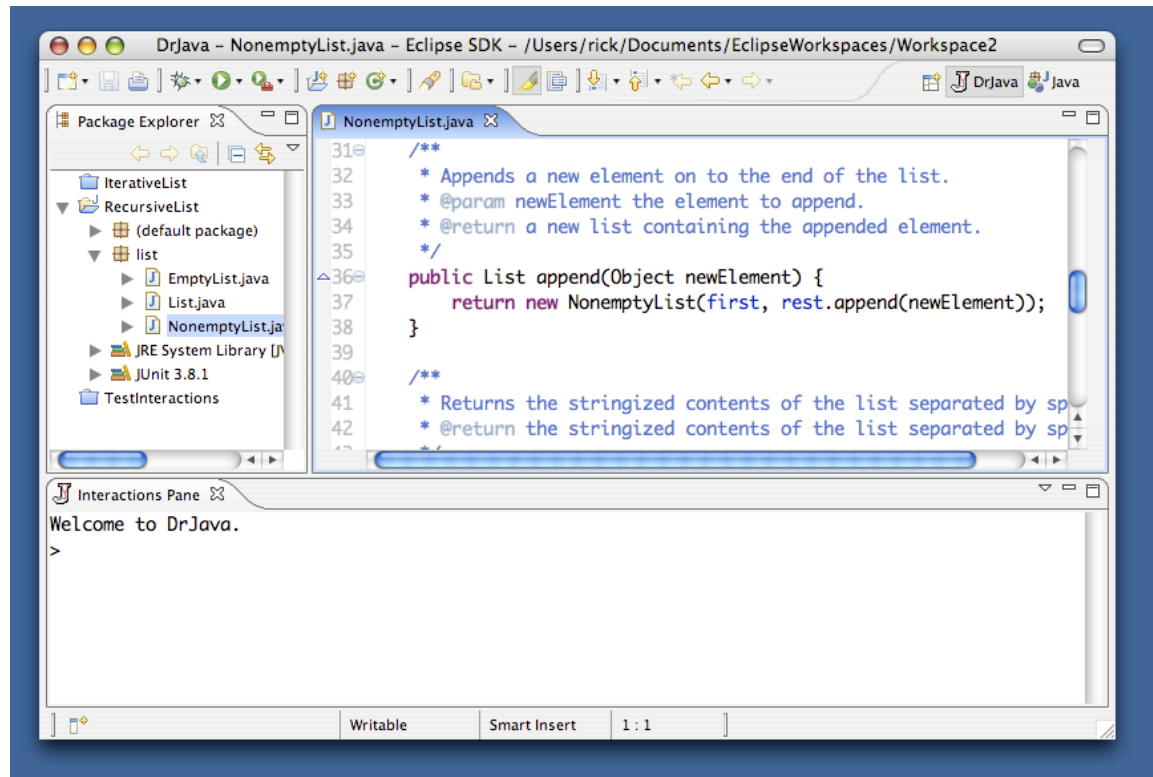


Figure 1.5: Eclipse. Shown here is a screenshot of the Eclipse integrated development environment with the DrJava plug in.

- Software can direct a processor to emulate another processor creating a virtual machine (VM).
- The Java platform uses virtual machine technology. A Java compiler translates Java source code into machine language for a Java virtual machine (JVM).
- Java is a modern language with features that support the demands of today's software such as security, networking, web deployment. Java is widely used commercially.
- An IDE is an integrated development environment—one program that provides all the tools that developers need to write software.
- DrJava is a simple IDE useful for beginning Java programmers.
- The DrJava interactive environment is available in both the DrJava IDE and Eclipse IDE.

1.6 Exercises

1. What advantages does Java's virtual machine concept provide compared to the traditional approach of compiling code for a particular platform? What are the disadvantages?
2. What is a compiler?
3. What is bytecode?
4. How is Java source code related to bytecode?
5. In DrJava, create a new file containing `RemainderTest` (1.1).

6. Compile the program and correct any typographical errors that may be present.
7. Run the program to verify that it correctly computes the remainder of $15 \div 6$.
8. Use the Interactions pane to directly evaluate the expression $15 / 6$.
9. Experiment more with the Interactions pane using numbers and the operators $+$, $-$, $*$, $/$, and $\%$. See if the use of parentheses makes any difference in the results. Reflect on the results of your activities and form some conclusions about how these operators work with numbers.
10. Using the web, research the similarities and differences between Java and Microsoft's C# programming language.
11. Using the web, research the similarities and differences between Java's runtime environment and Microsoft's .NET environment.
12. Visit DrJava's website (<http://www.drjava.org>) and begin becoming familiar with some of DrJava's features. How do its capabilities compare to other software applications you have used, such as a wordprocessor?

Chapter 2

Values, Variables, and Types

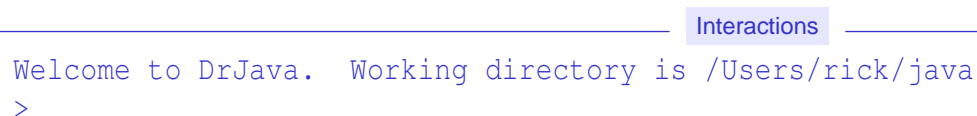
In this chapter we explore the primitive building blocks that are used to develop Java programs. We experiment with the following concepts within DrJava's interactive environment:

- numeric and nonnumeric values
- variables
- expressions

Next chapter we use these primitive concepts to build more sophisticated, computation-rich objects.

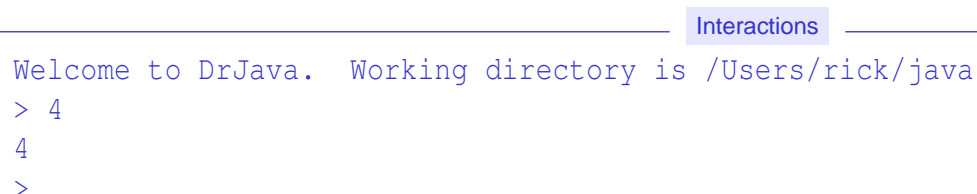
2.1 Java Values in DrJava's Interaction Pane

The DrJava environment provides an Interactions pane that is convenient for experimenting with Java's primitive types. When you select the **Interactions** tab in the lower window panel, you see



```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
>
```

The greater than (>) prompt indicates the Interactions interpreter is ready to receive input from the user. If you type the number four



```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> 4
4
>
```

the value four is displayed, and the prompt reappears waiting for additional input. In these simulated Interactions panes the user's input appears to the right of the prompt (>), and the response of the interpreter appears on a line without a prompt; this is exactly how it works in DrJava's Interactions pane. Also, when the message

```
Welcome to DrJava. Working directory is /Users/rick/java
```

appears in the Interactions pane it means that this is a fresh session. In a fresh session any prior user interactions are forgotten. A fresh session is instituted in one of several ways:

- A new interactive session is started when DrJava begins running.
- When the user issues the “Reset interactions” command from the DrJava menu the session is reinitialized.
- Compiling a Java source file within the Editor pane resets the Interactions pane.

The number four is an integer *value*. Java supports a variety of numeric value types and several kinds of nonnumeric values as well. Like mathematical integers, Java integers represent whole numbers, both positive and negative:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> -93
-93
```

Unlike mathematical integers, however, Java's integers have a limited range. Consider the following interactive session:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> 2147483646
2147483646
> 2147483647
2147483647
> 2147483648
NumberFormatException: For input string: "2147483648"
  at java.lang.NumberFormatException.forInputString(NumberFormat...
  at java.lang.Integer.parseInt(Integer.java:463)
  at java.lang.Integer.valueOf(Integer.java:553)
```

2,147,483,647 is the largest integer that can be represented by Java's “normal” integer type. Java's standard integer type is called `int`. This limited range is common among programming languages since each number is stored in a fixed amount of memory. Larger numbers require more storage in memory. In Java (as in many other languages), `ints` require four bytes (32 bits) of memory. 2,147,483,647 is the largest integer that can be stored in four bytes. In order to model the infinite set of mathematical integers an infinite amount of memory would be needed! As we will see later, Java supports an integer type with a greater range and also provides support for arbitrary-precision integers.

Now try to enter numbers with decimal places:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> 3.14159
3.14159
> -0.0045
-0.0045
```


These are not integers but are more like mathematical real numbers. The key here is “like.” The name of this decimal type is `double`, which stands for *double-precision floating point number*. The term *floating point* means that the decimal point can “float” so the number of digits behind the decimal point can vary. As the *double-precision* label implies, Java also supports single-precision floating point numbers.

Just like with `ints`, `doubles` use a fixed amount of memory, in this case eight bytes per value. This means that both the range and precision of `doubles` is limited. The largest double is

$$1.7976931348623157 \times 10^{308}$$

the smallest positive double is

$$4.9 \times 10^{-324}$$

`doubles` can be both positive and negative, and maintain a minimum of 15 digits of precision. Java's `doubles` are therefore only an approximation of mathematical real numbers. An irrational number like π cannot be represented exactly since π has an infinite number of digits. While integers can be represented exactly within the range of values, because of finite precision not all floating point values can be represented. Consider:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> 1.227835364547718468456
1.2278353645477185
```

The `double` type cannot store 22 decimal digits of precision, so the desired value is rounded to just 17 digits. Programmers must take care when performing complex calculations involving `doubles` to ensure that cumulative rounding errors do not lead to meaningless results.

Scientific notation can be used:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> 6.023e23
6.023E23
```

The number to the left of the `e` (capital `E` can be used as well) is the mantissa, and the number to the right of the `e` is the exponent of 10. `6.023e23` thus stands for 6.023×10^{23} .

One type of nonnumeric value is the `boolean` type. It has only two values: `true` and `false`. These values must be typed in exactly, as Java is case sensitive (capitalization matters):

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> true
true
> True
Error: Undefined class 'True'
> TRUE
Error: Undefined class 'TRUE'
> false
false
> False
Error: Undefined class 'False'
> FALSE
Error: Undefined class 'FALSE'
```


The word *Boolean* comes from George Boole, a mathematician that founded the algebra of mathematical logic. At first glance, the `boolean` type may appear rather limited and useless, but as we will see it is essential for building powerful programs.

Java supports other primitive data types. A comprehensive list can be found in Table 2.1, but we will have little need for them until later. For the time being we will restrict our attention to `ints`, `doubles`, and `booleans`.

Name	Meaning	Range	Size
<code>byte</code>	byte	$-128 \dots +127$	8 bits
<code>short</code>	short integer	$-32,768 \dots +32,767$	16 bits
<code>char</code>	Unicode character	$0 \dots +65,536$	16 bits
<code>int</code>	integer	$-2,147,483,648 \dots +2,147,483,647$	32 bits
<code>long</code>	long integer	$-9,223,372,036,854,775,808 \dots +9,223,372,036,854,775,807$	64 bits
<code>float</code>	single-precision floating point	$\pm 3.4 \times 10^{+38} \dots \pm 1.4 \times 10^{-45}$ with at least 7 decimal digits of precision	32 bits
<code>double</code>	double-precision floating point	$\pm 1.7 \times 10^{+308} \dots \pm 4.9 \times 10^{-324}$ with at least 15 decimal digits of precision	64 bits
<code>boolean</code>	Boolean	false or true	8 bits

Table 2.1: Primitive data types

One non-primitive type that is worth noting at this point is `String`. A string is a sequence of characters. In Java, string literals are enclosed within quotation marks:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> "This is a string"
"This is a string"
```

A string is an example of a Java object, and so it is not a primitive type. This means that strings have capabilities that exceed those of primitive types. Some of these capabilities will be explored later.

2.2 Variables and Assignment

In algebra, variables are used to represent numbers. The same is true in Java, except Java variables also can represent values other than numbers.

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> x = 5
5
> x
5
```

In an actual Java program the first two lines of the above interaction would be terminated with semicolons and be called *statements*. The semicolons can be used in the Interactions pane as well:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> x = 5;
> x
5
```

Notice how the terminating semicolon suppresses the evaluation of the entered expression; consider:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> 100
100
> 100;
>
```

The statement

```
x = 5;
```

is called an *assignment statement* because it assigns a value to a variable. The `=` operator is called the *assignment operator*, and its meaning is different from equality in mathematics. In mathematics, `=` asserts that the expression on its left is equal to the expression on its right. In Java, `=` makes the variable on its left take on the value of the expression on its right. It is best to read `x = 5` as “`x` is assigned the value 5,” or “`x` gets the value 5.” This distinction is important since in mathematics equality is symmetric: if $x = 5$, we know $5 = x$. In Java, this symmetry does not exist:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> x = 5
5
> 5 = x
Error: Bad left expression in assignment
```

The command `5 = x` attempts to reassign the value of the literal integer value 5, but this cannot be done because 5 is always 5 and cannot be changed. To further illustrate the point consider:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> x = 5;
> x
5
> x = x + 1;
> x
6
```

In mathematics no number satisfies the equation $x = x + 1$. In Java the statement `x = x + 1` works as follows:

- The expression to the right of the `=` operator, `x + 1`, is evaluated. The current value of `x` (5 in this case) is added to 1, and so the right-hand side of the `=` evaluates to 6.

- The value 6 is then assigned to `x`.

Variables can be reassigned different values as needed:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 10;
> x
10
> x = 20;
> x
20
```

Notice a variable may not be declared twice in the same interactive session.

Variables may be of type `double`, `boolean`, and `String`:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> amount = 12.08;
> fact = true;
> name = "Lance";
> amount
12.08
> fact
true
> name
"Lance"
```

While unnecessary in DrJava's Interactions pane, in a Java program variables must be declared before they are used. A declaration specifies a variable's type to the compiler. The compiler will issue an error if a programmer attempts to use an undeclared variable in a Java program. Variables can optionally be declared in an Interactions pane:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x;
> x = 5;
> x
5
```

The statement

```
int x;
```

is a *declaration statement*. We saw how variables may be assigned different values as often as we like, but a given variable may be declared only once in a given context:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 10;
```

```
> x
10
> x = 20;
> x
20
> int x;
Error: Redefinition of 'x'
```

A variable may not be declared twice in the same interactive session.

Variables of type `double`, `boolean`, and `String` can be declared interactively and *must* be declared within a Java program:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> double amount = 12.08;
> boolean fact = true;
> String name = "Lance";
> amount
12.08
> fact
true
> name
"Lance"
```

Note that even though we use the term *string* in a generic sense, the type name for string variables is `String` with a capital S.

Sometimes programmers need to have numeric objects with immutable values; that is, their values cannot be changed. Such *constants* are specified by the keyword `final`:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> final int C = 100
> C
100
> C = 45
Error: This object cannot be modified
```

Here `C` is a constant that may not be modified. We say that `C` is *read only*; once its value has been specified you may look at and use its value, but you may not change its value. This notion of using names for unvarying values is common in mathematics and the sciences; for example, $\pi \approx 3.14159$ (ratio of a circle's circumference to its diameter) and $c = 2.998 \times 10^8 \frac{\text{m}}{\text{sec}}$ (speed of light). While not required by the language, the convention in Java is to use all capital letters for constant names. These symbolic constants can make Java source code more readable, as in:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> final double PI = 3.14159
> PI
3.14159
```

Here π stands for the mathematical constant π .

Table 2.2 shows various ways in which declaration statements can be written. Whether in a Java program or

Description	General Form	Example
Declare one variable and do not give it an initial value	<i>type variable;</i>	<code>int x;</code>
Declare one variable and give it an initial value	<i>type variable = value;</i>	<code>int x = 5;</code>
Declare multiple variables of the same type and give none of them initial values	<i>type variable₁, variable₂, ..., variable_n;</i>	<code>int x, sum, height;</code>
Declare multiple variables of the same type and give some or all of them initial values	<i>type variable₁ = value₁, variable₂ = value₂, ..., variable_n = value_n;</i>	<code>int x = 5, sum, height = 0;</code>

Table 2.2: Structure of declaration statements. Commas separate variables, and the statement ends with a semicolon.

in the Interactions pane, a variable may not change its type during its lifetime. When an undeclared variable is used within the Interactions environment DrJava automatically determines the variable's type based on the type of the expression on the right-hand side of the assignment operator. Just as in Java programs, though, during a given interactive session a variable's type is fixed:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> y = 19;
> y
19
> y = 12.4;
Error: Bad types in assignment
```

Here the interpreter correctly determined that `y` has type `int` when it assigned 19 to `x`. An attempt to assign a double value to `y` then resulted in an error since it is illegal to assign a double value to an `int` variable. Notice that the opposite assignment is acceptable:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> z = 12.4;
> z
12.4
> z = 19;
```

Why is this so? Remember that the range of doubles is much larger than the range of ints. In fact, any int value can be represented as a double, so it makes sense to automatically convert an int value to a double for the purposes of assignment. The conversion in the other direction has potential problems because not all double values can be represented as ints because most legal double values fall outside the range of ints. The compiler (and interpreter) thus prohibit the automatic assignment of a double value to an int variable.

What if the double value is in the range of ints? The programmer can force a double value to be assigned to an int variable through an *explicit cast*. Consider

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> int x;
> x = (int) 12.7;
> x
12
```

The name of the type, in this case `int`, is placed in parentheses before the expression to convert. Notice that the value is truncated; this means the decimal digits are simply dropped and the value is not rounded properly. The programmer must be careful, though:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> int x;
> x = (int) 4.29e15;
> x
2147483647
```

The value 4.29×10^{15} is a legitimate mathematical integer, but it is much too big to be a Java `int`. Notice that the compiler does the best it can under the circumstances—it gives `x` the biggest value it can.

A boolean value may not be assigned to a numeric type and vice-versa. Similarly, strings are assignment incompatible with numbers and boolean values.

2.3 Identifiers

Java has strict rules for variable names. A variable name is one example of an *identifier*. An identifier is a word that can be used to name a variable. (As we will see in later chapters, identifiers are also used to name methods and classes.) Identifiers can be created as follows:

- Identifiers must contain at least one character
- The first character must be an alphabetic letter (upper or lower case), the underscore, or the dollar sign

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_\$

- The remaining characters (if any) may be alphabetic characters (upper or lower case), the underscore, the dollar sign, or a digit

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_\$0123456789

- No other characters (including spaces) are permitted in identifiers
- A reserved word cannot be used as an identifier

Java reserves a number of words for special use that could otherwise be used as identifiers. Called *reserved words*, these words are used to define the structure of Java programs and statements. Table 2.3 lists all the Java reserved words.

abstract	else	interface	super
boolean	extends	long	switch
break	false	native	synchronized
byte	final	new	this
case	finally	null	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	true
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while
double	int	strictfp	

Table 2.3: Java reserved words

In most programming languages the terms *reserved word* and *keyword* are used interchangeably; in Java, a keyword is a reserved word that has special meaning when describing the structure of a program [2]. All of the *reserved words* in Table 2.3 are keywords except `null`, `true`, and `false`. These three reserved words are technically classified as special literal values, not keywords. None of the reserved keys in Table 2.3 can be used as identifiers. Fortunately, if you accidentally use one of the reserved words in Table 2.3 as a variable name, the compiler (and the interactions pane) will issue an error:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> void = 11
Syntax Error: "void ="
```

(§ 8.5). The purposes of many of these reserved words are revealed throughout this book.

While mathematicians are content with giving their variables one-letter names (like x), programmers should use longer, more descriptive variable names. Names such as `sum`, `height`, and `subTotal` are much better than the equally permissible `s`, `h`, and `st`. A variable's name should be related to its purpose within the program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols quite understandable.

Java is a case-sensitive language. This means that capitalization matters. `if` is a reserved word, but none of `If`, `IF`, or `iF` are reserved words. Identifiers are case sensitive also:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> String name = "Clyde";
```

```
> Name  
Error: Undefined class 'Name'
```

The variable `name` was declared in all lowercase letters, so the variable `Name` does not exist in this session.

2.4 Summary

- Java supports a variety of numeric types.
- Numbers represented on a computer have limitations based on the finite nature of computer systems.
- Nonnumeric types include Boolean values, strings, and other object types.
- Variables are used to store values.
- In a Java program, all variables *must* be declared; in DrJava's Interactions pane, variables *may* be declared.
- Variables are assigned values in assignment statements using the `=` operator.
- Variable names follow the rules for naming identifiers as indicated in § 2.3.
- Constants are defined with the `final` specifier.
- Reserved words cannot be used as variable names.
- Floating point values can be assigned to integer variables via a casting operation. Programmers must be careful using casts because it is easy to introduce errors into a program via a legal but bad cast.

2.5 Exercises

1. How does DrJava's Interactions Pane respond if you enter a number? (For example, type 35 and press **Enter**.)
2. How do integer values differ from floating point values?
3. What is Java's "standard" integer type?
4. What is Java's "standard" floating-point type?
5. How many primitive types does Java support? List them.
6. What is the largest `int` value supported in Java?
7. What is the largest floating point value supported by Java's built-in floating point types?
8. How, in Java source code, can you represent the the largest floating point value supported by Java's built-in floating point types?
9. List all possible `boolean` values.
10. Explain how Java's `=` operator differs from from the same symbol used in mathematics. Provide one legitimate example from mathematics that would be an illegal Java expression.
11. How can can you declare the variable named `total` to be type integer?

12. What is a string? Is a string a primitive type?
13. What does the `final` specifier do when declaring a variable?
14. In our example, why did we declare `PI` to be `final`?
15. In our example, why did we use the name `PI` instead `pi`, `Pi`, or `pI`? Could we have used those names instead?
16. How is DrJava's Interaction's Pane difference from a Java program in terms of variable declarations?
17. What is the difference between a variable name and an identifier?
18. Classify each of the following as either a *legal* or *illegal* Java identifier:

<code>fred</code>	<code>if</code>
<code>sum_total</code>	<code>While</code>
<code>sumTotal</code>	<code>\$16</code>
<code>sum-total</code>	<code>_4</code>
<code>sum total</code>	<code>_____</code>

19. What can you do if a variable name you would like to use is the same as a reserved word?

Chapter 3

Arithmetic Expressions

Java programs can solve mathematical problems. Often programs that appear nonmathematical on the surface perform a large number of mathematical calculations behind the scenes. Graphical computer games are one example. The user experience with a game may be very different from that of solving algebra or trigonometry problems, but the executing program is continuously doing the mathematics behind the scenes in order to achieve the images displayed on the screen. This chapter examines some of Java's simpler mathematical capabilities.

3.1 Expressions and Mathematical Operators

A literal value (like 5) and a variable (like *x*) are simple *expressions*. Complex expressions can be built from simpler expressions using *operators*. Java supports the familiar arithmetic operators as show in Table 3.1. The arithmetic

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder)

Table 3.1: Arithmetic operators

operators show in Table 3.1 are called *binary operators* because they require two operands. The following interactive session illustrates addition:

Interactions

```

Welcome to DrJava.  Working directory is /Users/rick/java
> 2 + 2
4
> 10+4
14
> int x = 5;
> x + 3
8

```

Space around the `+` operator is optional but encouraged since it makes the expression easier to read.

The following session computes the circumference of a circle with a radius of 10 using the formula $C = 2\pi r$:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
```

π is a mathematical constant and should not vary; therefore, `PI` is declared `final` so its value cannot be changed accidentally later on during the session.

It is important to realize that the statement

```
double circumference = 2 * PI * radius;
```

is not definition of fact, but simply a computation and an assignment that is executed when encountered. To illustrate this concept, consider the following extended session:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
> radius = 4;
> circumference
62.8318
```

Here we see how even though `radius` changed, `circumference` was not automatically recomputed. This is because after `radius` was reassigned, `circumference` was *not* reassigned. Therefore, `circumference` must be recomputed if `radius` changes:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
> radius = 4;
> circumference = 2 * PI * radius;
> circumference
25.13272
```

As in algebra, multiplication and division have precedence over addition and subtraction, and parentheses can be used for grouping operations and overriding normal precedences.

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> 2 * 3 + 5
11
> 2 + 3 * 5
17
> 2 * (3 + 5)
16
> (2 + 3) * 5
25
```

In algebra, it is common to write the multiplication of two variables, $x \times y$, as xy . Mathematicians can do this because they use one letter variable names. Java allows and encourages longer variable names, so the multiplication operator ($*$) may not be omitted because if you have variables named x and y , xy is a new variable name. x times y must be written as $x * y$.

It may seem odd to include a remainder operator, but it actually quite useful. Consider the problem of computing the number of hours, minutes, and seconds in a given number of seconds. For example, how many hours, minutes, and seconds are there in 10,000 seconds? First let us develop some basic identities:

- 60 seconds = 1 minute
- 60 minutes = 1 hour

This means that one hour has 60×60 seconds, or 3,600 seconds. Armed with this information, we can do some calculations in the DrJava interpreter:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> int sec = 10000;
> int hr = sec/3600;
> sec = sec % 3600;
> int min = sec/60;
> sec = sec % 60;
> hr
2
> min
46
> sec
40
```

Thus 10,000 seconds equals 2 hours, 46 minutes, 40 seconds. Here is how the calculation works:

- `int sec = 10000;`
- `int hr = sec/3600;`

Set the number of seconds to 10,000, the starting point of our computation.

Since each hour has 3,600 seconds, we divide `sec` by 3,600. Note that Java you may not use commas to separate groups of digits. The result is assigned to the variable `hr`.

- `sec = sec % 3600;`

Here we reassign `sec` to be the number of seconds remaining after the previous calculation that computed the hours. The previous calculation assigned `hr` but did not modify `sec`. This assignment statement actually modifies `sec`.

- `int min = sec/60;`
`sec = sec % 60;`

Similar to the previous item, we compute the number of minutes from the remaining seconds and then find the remaining seconds after the minutes computation, updating the seconds.

We can easily check our answer:

Interactions

```
> 2*60*60 + 46*60 + 40
10000
```

Java has two *unary* arithmetic operators. They are called unary operators because they have only one operand. The unary `+` has no effect on its operand, while unary `-` returns the additive inverse of its operand:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> 2 + 3 * 4
14
> +(2 + 3 * 4)
14
> -(2 + 3 * 4)
-14
1
> x = 5;
> -x
-5
```

3.2 String Concatenation

Strings can be spliced together (a process known as *concatenation*) using the `+` operator. Consider:

Interactions

```
> "Part 1" + "Part2"
"Part 1Part2"
```

The compiler and interpreter will automatically convert a primitive type into a human-readable string when required; for example, when a string is concatenated with a primitive type:

Interactions

```
> "Part " + 1
"Part 1"
> "That is " + true
"That is true"
```

Notice that neither `1` nor `true` are strings but are, respectively, `int` and `boolean` literals. We say the `+` operator is *overloaded* since it performs two distinct functions:

- When both operands are numbers, `+` performs familiar arithmetic addition.
- When one or both of its operands is a `String`, `+` converts the non-`String` operand to a `String` if necessary and then performs string concatenation.

String variables can be concatenated just like string literals:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> String phrase = "The time is now";
> int hour = 4, minute = 12;
> String time = phrase + " " + hour + ":" + minute + " exactly.";
> time
"The time is now 4:12 exactly."
```

3.3 Arithmetic Abbreviations

Expressions do not affect the value of variables they contain. The expression

$$x + 1$$

evaluates to one more than `x`'s value, but `x`'s value is not changed. It is misleading to say the expression “adds one to `x`,” since this implies `x` becomes one more than before the expression was evaluated. An assignment operator actually changes a variable's value. We have seen the simple assignment operator, `=`. Several more assignment operators exist that allow arithmetic to be combined with assignment. The statements

```
x = x + 1;
y = y * 2;
z = z - 3;
w = w / 4;
t = t % 5;
```

can all be expressed in a different, shorter way:

```
x += 1;
y *= 2;
z -= 3;
w /= 4;
t %= 5;
```

In all the arithmetic-assignment operators, no space can come between the arithmetic symbol and the equals symbol.

The task of increasing or decreasing a variable's value is common in computing. Java provides the following shortcut:

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

can be rewritten as:

```
x++;
```

or as:

```
++x;
```

There is a slight different between the two notations as an exercise illustrates. When used as a standalone statements as shown here, the preincrement (`++x`) and postincrement (`x++`) versions work the same. Similarly,

```
x--;
```

and

```
--x;
```

decrease `x`'s value by one. Note how the `++` and `--` operators *are* assignment operators even though no `=` symbol appears in the operator. As with the arithmetic-assignment operators, no space may appear between the symbols that make up the `++` and `--` operators. The `++` and `--` variables can only be applied to numeric variables, not expressions and not nonnumeric variables.

3.4 Summary

- The arithmetic operators `+`, `-`, `*`, `/`, and `%` perform mathematical operations on variables and values.
- In the expression `x + y`, `+` is the operator, while `x` and `y` are its operands.
- The unary `+` and `-` require only a single operand.
- Arithmetic expressions mixing `+`, `-`, `*`, and/or `/` are evaluated in the same order as they are in normal arithmetic (multiplication before addition, etc.).
- As in standard arithmetic, parentheses can be used to override the normal order of evaluation.
- The `+` operator can be used to concatenate two strings. Any nonstring operands are converted into strings.
- Java's statements involving assignment with simple arithmetic modification have convenient abbreviations.

3.5 Exercises

1. How does the `/` operator differ from the `÷` operator? Provide an example illustrating the difference.
2. Evaluate each of the following Java expressions:

- | | | | | |
|---------------------------|-------------------------|-----------------------------|-----------------------------|---------------------------|
| a. <code>2 + 2</code> | b. <code>2 % 2</code> | c. <code>5 / 3</code> | d. <code>5 % 3</code> | e. <code>5.0 / 3.0</code> |
| f. <code>5.0 / 3</code> | g. <code>5 / 3.0</code> | h. <code>(1 + 2) * 5</code> | i. <code>(1 * 2) + 5</code> | j. <code>1 + 2 * 5</code> |
| k. <code>1 * 2 + 5</code> | l. <code>--4</code> | m. | n. | o. |

3. Is it legal for the same variable name to appear on both sides of an assignment statement? How is this useful? What is the restriction on the left-hand side of the statement?
4. How is an expression different from a statement?
5. What is concatenation? On what type is concatenation performed?
6. What happens when a string is concatenated with an `int` value? What happens when a string is concatenated with an `int` variable?
7. How are unary operators different from binary operators?
8. If necessary consult a mathematics book to review the basic properties of arithmetic operations, then answer the following questions about Java's arithmetic operations:
 - a. Is `+` commutative over `ints`?
 - b. Is `+` associative over `ints`?
 - c. Is `*` commutative over `ints`?
 - f. Does the distributive property hold for `*` and `+`?
 - g. Is `/` closed over `ints`?
 - h.
 - k.
 - l.
 - m.
9. What is the difference in behavior of the following two statements:

```
x += 2;  
x =+ 2;
```

and the following two statements:

```
x -= 5;  
x =- 5;
```

and the following two statements:

```
x *= 3;  
x =* 3;
```

Explain the results.

10. Set up the situation where you can use the following statements in the interpreter:

```
y = x++;
```

and

```
y = ++x;
```

Explain why it really does make a difference whether the `++` comes before or after the variable being incremented.

Does your theory hold for the `--` operator?

Chapter 4

Objects: Packaging Computation

Java is an *object-oriented language*. This means useful work is done through computational elements called *objects*. A variable is an instance of a type, such as an `int`, `double`, or `boolean`. These built-in types are called *primitive types* because they provide simply a value. An object, which also may be represented by a variable, is an instance of a programmer-defined type. A programmer-defined type can specify both value and behavior.

4.1 Classes

A programmer defines the characteristics of a type of object through a *class*. The term *class* as used here means *class of objects*. A class is a template or pattern that defines the structure and capabilities of an object. All objects that are members of the same class share many common characteristics. A class can define both *data* and *behavior*:

- data—objects can store primitive values (`ints`, `booleans`, etc.), as well as references to other objects
- behavior—objects can specify operations that produce useful results to the user of the object

Initially we will define simple classes that allow us to create an object that can be used as a miniprogram. While a class can be defined within the DrJava interpreter, it is more convenient to use the editor (also known as the Definitions pane). Recall from geometry the formulas for the circumference and area of a circle given its radius. Figure 4.1 illustrates.

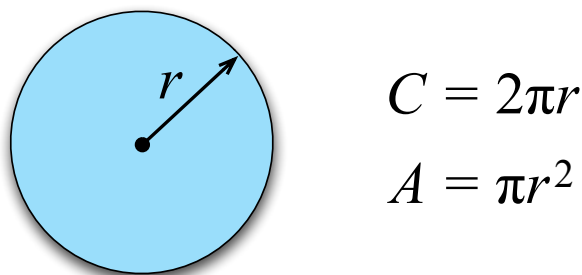


Figure 4.1: Formulas for the circumference and area of a circle given its radius: $C = 2\pi r$ and $A = \pi r^2$

Enter the code found in CircleCalculator (Figure 4.1) into the DrJava editor and save it as CircleCalculator. DrJava will then create a file named CircleCalculator.java that stores the source code you type in.

```
public class CircleCalculator {  
    private final double PI = 3.14159;  
    public double circumference(double radius) {  
        return 2 * PI * radius;  
    }  
    public double area(double radius) {  
        return PI * radius * radius;  
    }  
}
```

Listing 4.1: CircleCalculator—a class used to create CircleCalculator objects that can compute the circumference and area of circles

In this CircleCalculator class:

- Class declaration:

```
public class CircleCalculator {
```

The reserved word `class` signifies that we are defining a new class named CircleCalculator. The reserved word `public` when used before `class` indicates that the class is meant for widespread use. Most of the classes we consider will be public classes. The open curly brace (`{`) marks the beginning of the class body.

- Attribute declaration:

```
    private final double PI = 3.14159;
```

This declaration means that every CircleCalculator object created will have a double-precision floating point constant named `PI`. The reserved word `private` indicates that this constant is only available to code within this class; it is not accessible to code outside of this class. It is a constant since it is declared `final`, and so it will always have the value of 3.14159. `PI` is created and initialized when the object itself is created.

- Method declaration:

```
    public double circumference(double radius) {
```

The parentheses following the name `circumference` indicate that this is a *method declaration*. A *method* is a named collection of Java statements that can be executed as a miniprogram.

- The `public` specifier means any code using CircleCalculator objects can invoke this method.
- The `double` specifier before its name indicates the type that this method will return. A `return` statement must appear somewhere within this method's body, and the `return` statement must have an associated expression assignment-compatible with `double`.
- This method's name is `circumference`. Clients that use objects of this class will *call*, or *invoke*, this method by its name.

- Parentheses surround the method's *parameter list*, also known as its *argument list*. Clients are required to pass information to the method when its parameter list is not empty. Here, a client must provide a single `double` value when it calls this method. This parameter named `radius` is a variable that is used within the method body to participate in the computation.
- The open curly brace marks the beginning of the *method body*.

Unlike some other programming languages, Java does not allow a method to be defined outside the context of a class.

- Method body:

```
return 2 * PI * radius;
```

A method's body consists of Java statements. Unlike in the DrJava interpreter, all Java statements must be terminated with a semicolon. This method contains only one statement which computes the product of three numbers:

- 2, a literal constant,
- `PI`, the symbolic, programmer-defined constant mentioned above, and
- `radius`, the parameter—the caller provides the value of `radius` as we will see shortly.

The method returns the result of this expression to the caller.

- End of method body:

```
}
```

The close curly brace after the `return` statement marks the end of the method's body. It matches the open curly brace that marks the beginning of the method's body.

- `area()` is a method similar to `circumference()`.
- End of class:

```
}
```

The last close curly brace terminates the class definition. It matches the open curly brace that marks the beginning of the class definition. Curly braces with a Java class must be properly nested like parentheses within an arithmetic expression must be properly nested. Notice that the curly braces of the methods properly nest within the curly braces of the class definition.

In order to use this `CircleCalculator` class we must compile the source code into executable bytecode. DrJava has three ways to compile the code:

- select the **Compile** button in the toolbar,
- from the main menu, select *Tools*, then *Compile current document*, or
- press the **Shift F5** key combination.

The Compiler Output pane should report

Compiler Output

Compilation completed.

If the compiler pane indicates errors, fix the typographical errors in the source code and try again. Once it compiles successfully, we can experiment with `CircleCalculator` objects in the DrJava interpreter:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> CircleCalculator c = new CircleCalculator();
> c
CircleCalculator@e8e5a7
> c.circumference(10.5)
65.97339
> c.area(10.5)
346.3602975
> x = c.circumference(10.5);
> x
65.97339
> c.PI
IllegalAccessException: Class
koala.dynamicjava.interpreter.EvaluationVisitor can not access a
member of class CircleCalculator with modifiers "private final"
at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
at java.lang.reflect.Field.doSecurityCheck(Field.java:954)
at java.lang.reflect.Field.getFieldAccessor(Field.java:895)
at java.lang.reflect.Field.get(Field.java:357)
> c.PI = 2.5;
Error: This object cannot be modified
```

Let us analyze what transpired during this interactive session:

- First we created a `CircleCalculator` object, or instance, named `c`.

```
> CircleCalculator c = new CircleCalculator();
```

The declared type of variable `c` is `CircleCalculator`. The variable `c` here represents a reference to an *object*. The terms *object* and *instance* are interchangeable. The reserved word `new` is used to create an object from a specified class. Although it is a word instead of a symbol (like `+` or `*`), `new` is technically classified as an operator. The word following the `new` operator must be a valid type name. The parentheses after `CircleCalculator` are required. This first line is a Java declaration and initialization statement that could appear in a real Java program.

While `c` technically is a reference to an object, we often speak in less formal terms, calling `c` itself an object. The next item highlights why there truly is a distinction between an object and a reference to that object.

- Next, we evaluated the variable `c`.

```
> c
CircleCalculator@e8e5a7
```

Note the somewhat cryptic result. Later (§ 14.2) we will cover a technique to make our objects respond in a more attractive way. By default, when the interpreter evaluates an object of a class we create, it prints the object's class name, followed by the @ symbol, and then a number in hexadecimal (base 16) format. Hexadecimal numbers use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. You do not need to know the hexadecimal number system for our purposes; simply be aware that the Java runtime environment (JRE) maps the number to an address in the computer's memory. We can view this number as a unique ID or serial number for the object.¹

The variable itself thus holds the address of an object, not the contents of the object itself. That is why we properly call `c` a reference to a `CircleCalculator` object. Again, often we will be less formal, calling `c` a `CircleCalculator` object.

What can we do with object `c`? `c` is a `CircleCalculator` object, so we can use `c` to calculate the circumference and area of circles.

- Once we have created the object `c` we use the `circumference()` method to compute the circumference of a circle with the radius 10.5.

```
> c.circumference(10.5)
65.97339
```

We say we are “calling” or “invoking” the `circumference()` method of the `CircleCalculator` class on behalf of object `c` and passing to it the radius 10.5. In order to call a method on behalf of an object, we write the name of the object (in this case `c`), followed by the dot operator (`.`), followed by the method's name, and then a set of parentheses:

object . method ()

Within the Interactions environment we are a *client* of object `c` and a *caller* of its `circumference()` method.

- Next, we call the `area()` method to compute the area of a circle with the radius 10.5.

```
> c.area(10.5)
346.3602975
```

- We can assign the result of calling the method to a variable if we like:

```
double x = c.circumference(10.5);
```

- Finally, we attempt to use `c`'s `PI` constant. Since it is declared `private`, the interpreter generates an error.

```
> c.PI
IllegalAccessException: Class
koala.dynamicjava.interpreter.EvaluationVisitor can not access a
member of class CircleCalculator with modifiers "private final"
at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
at java.lang.reflect.Field.doSecurityCheck(Field.java:954)
at java.lang.reflect.Field.getFieldAccessor(Field.java:895)
at java.lang.reflect.Field.get(Field.java:357)
```

Since we cannot look at `c.PI` we certainly cannot modify it:

¹Why is hexadecimal numbering used? Hexadecimal numbers are commonly used for specifying memory addresses because they are easily converted to binary (base 2) numbers and they can be expressed in fewer digits than their decimal (base 10) counterparts.

```
> c.PI = 2.5;
Error: This object cannot be modified
```

As an experiment, see what happens if you change the declaration of `PI` to be `public` instead of `private`. Next, remove the `final` specifier and try the above interactive sequence again.

Now that we have a feel for how things work, let us review our `CircleCalculator` class definition. A typical class defines *attributes* and *operations*. `CircleCalculator` specifies one attribute, `PI`, and two operations, `circumference()` and `area()`. An attribute is also called a *field* or *instance variable*. In the case of `CircleCalculator`, our instance “variable” is really a constant because it is declared `final`. A more common name for an operation is *method*. A method is distinguished from an instance variable by the parentheses that follow its name. These parentheses delimit the list of parameters that the method accepts in order to perform a computation. Both `circumference()` and `area()` each accept a single `double` value as a *parameter*, also known as an *argument*. Notice that the type of the parameter must be specified in the method’s parameter list. The parameter has a name so that it can be used within the method’s body.

A method may be written to accept as many parameters as needed, including none. To see how multiple parameters are used consider the task of computing the perimeter and area of a rectangle as shown in Figure 4.2. Whereas a

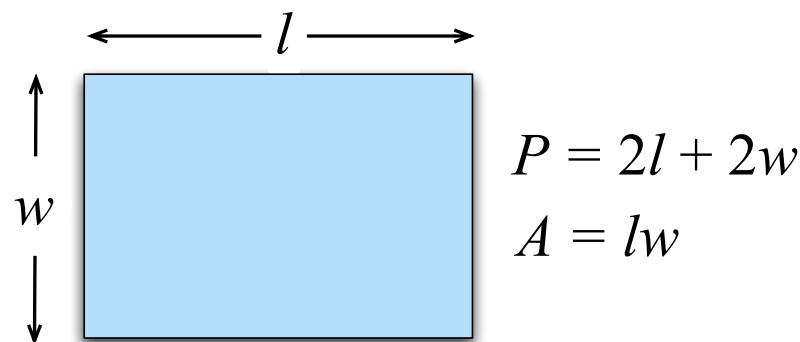


Figure 4.2: Formulas for the perimeter and area of a rectangle given its length and width: $P = 2l + 2w$ and $A = lw$.

a circle’s perimeter and area can be computed from only one piece of information (its radius), two pieces of information are required to compute the perimeter and area of rectangles—length and width. If a method requires multiple pieces of information to do its job, multiple parameters are necessary. (Do not confuse the words *parameter* and *perimeter*!) Multiple parameters are separated by commas, as shown in `RectangleCalculator` (Figure 4.2):

```
public class RectangleCalculator {
    public double area(double length, double width) {
        return length * width;
    }
    public double perimeter(double length, double width) {
        return 2 * length + 2 * width;
    }
}
```

Listing 4.2: `RectangleCalculator`—a class used to create `RectangleCalculator` objects that can compute the areas and perimeters of rectangles

The types of each parameter must be specified separately as shown. Here, both parameters are of type `double`, but in general the parameter types may be different.

In the interpreter we are the clients of `CircleCalculator` objects and `RectangleCalculator` objects. As we will soon see, code that we or others may write can also use our `CircleCalculator` and `RectangleCalculator` classes to create and use `CircleCalculator` and `RectangleCalculator` objects. We refer to such code as *client code*.

The return type of a method must be specified in its definition. The value that a method may return can be of any legal Java type. If a variable can be declared to be a particular type, a method can be defined to return that type. In addition, a method can be declared to “return” type `void`. A `void` method does not return a value to the client. Based on what we have seen so far, `void` methods may appear to be useless, but in fact they are quite common and we will write a number of `void` methods. While `void` is a legal return type for methods, it is illegal to declare a variable to be type `void`.

Methods get information from clients via their parameter lists and return information to clients via their return statement(s). An empty parameter list (a pair of parentheses with nothing inside) indicates that the method can perform its task with no input from the client. If a client tries to pass information to a method defined to not accept any parameters, the compiler will issue an error. Likewise, any attempt by a client to pass the wrong kinds of parameters to a method will be thwarted by the compiler.

A method has exactly one definition, but it can be invoked many times (even none).

4.2 Comments

Good programmers annotate their code. They insert remarks that explain the purpose of a method or why they chose to write a section of code the way they did. These notes are meant for human readers, not the compiler. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identifiers (§ 2.3) and comments can aid this process, especially when the comments disagree with the code! Also, in practice, teams of programmers develop software. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Java supports three types of comments:

- single line comments
- block comments
- documentation comments

We’ll describe the first two kinds of comments here; the documentation comments, which we will not cover here, are used to annotate the source code in such a way so that a special tool, `javadoc`, can create external documentation that can be read with any web browser.

Any text contained within comments is ignored by the compiler and the DrJava interpreter. Comments do not become part of the compiled bytecode, so providing generous comments does not affect the size or efficiency of the finished program in any way.

The first type of comment is useful for writing a single line remark:


```
> // Compute the circumference
> double circumference = 2 * PI * radius;
```

Here, the comment explains what the statement that follows it is supposed to do. The comment begins with the double forward slash symbols (//) and continues until the end of that line. The compiler and interpreter will ignore the // symbols and the contents of the rest of the line. This type of comment is also useful for appending a short comment to the end of a statement:

```
> count = 0; // Reset counter for new test
```

Here, an executable statement and the comment appear on the same line. The compiler or interpreter will read the assignment statement here, but it will ignore the comment. This is similar to the preceding example, but uses one line of source code instead of two.

The second type of comment is begun with the symbols /* and is in effect until the */ symbols are encountered. The /* . . . */ symbols delimit the comment like parentheses delimit a parenthetical expression. Unlike parentheses, however, these block comments cannot be nested within other block comments. The block comment is handy for multi-line comments:

```
> /* Now we are ready to compute the
   circumference of the circle.. */
> double circumference = 2 * PI * radius;
```

What should be commented? Avoid making a remark about the obvious; for example:

```
result = 0; // Assign the value zero to the variable named result
```

The effect of this statement is clear to anyone with even minimal Java programming experience. Thus, the audience of the comments should be taken into account. Generally, “routine” activities require no remarks. Even though the *effect* of the above statement is clear, its *purpose* may need a comment. For example:

```
result = 0; // Ensures 'result' has a well-defined minimum value
```

This remark may be crucial for readers to completely understand how a particular part of a program works. In general, programmers are not prone to providing too many comments. When in doubt, add a remark. The extra time it takes to write good comments is well worth the effort.

4.3 Local Variables

TimeConverter (Figure 4.3) conveniently packages into a class the seconds to hours:minutes:seconds conversion from Chapter 2. Its `convert()` method returns a string separating the hours, minutes, and seconds:

```
public class TimeConverter {
    public String convert(int seconds) {
        // First compute hours, minutes, and seconds
        int hours = seconds/3600;
        seconds = seconds % 3600;
        int minutes = seconds/60;
```



```

        seconds = seconds % 60;
        // Next, construct and return the string with the results
        return hours + " hr, " + minutes + " min, " + seconds + " sec";
    }
}

```

Listing 4.3: TimeConverter—converts seconds to hours, minutes, and seconds

In TimeConverter (Figure 4.3) the `convert()` method works as follows:

- The first part duplicates the work we did in an earlier interactive session, albeit with longer variable names:

```

int hours = seconds/3600;
seconds = seconds % 3600;
int minutes = seconds/60;
seconds = seconds % 60;

```

The number of hours, minutes, and seconds are calculated.

- The second part simply assembles a string containing the results and returns the string. The string concatenation operator (+) works nicely to build our custom string combining known labels (*hr*, *min*, and *sec*) with to-be-determined values to put with these labels.

As shown in this Interactions session, the method works as desired:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> t = new TimeConverter();
> t.convert(10000)
"2 hr, 46 min, 40 sec"
> t.convert(10025)
"2 hr, 47 min, 5 sec"
> t.convert(0)
"0 hr, 0 min, 0 sec"
> t.convert(1)
"0 hr, 0 min, 1 sec"
> t.convert(60)
"0 hr, 1 min, 0 sec"

```

The variable `seconds` is a parameter to the method. The variables `hours` and `minutes` are declared and used *within* the `convert()` method. This means these particular variables can only be used within the `convert()` method. We say they are *local* to `convert()`, and they are called *local variables*. If `TimeConverter` had any other methods, these local variables would not be available to the other methods. Local variables declared within other methods would not be available to `convert()`. In fact, another method could declare its own local variable named `hours`, and its `hours` variable would be different from the `hours` variable within `convert()`. Local variables are useful because they can be used without fear of disturbing code in other methods. Compare a local variable to an instance variable like `PI` from `CircleCalculator` (Figure 4.1). `PI` is declared outside of both the `circumference()` and `area()` methods, and it can be used freely by both methods. The variable `hours` in `TimeConverter` (Figure 4.3) is declared inside the `convert()` method, not outside, and so it can only be used by statements within `convert()`.

Unlike instance variables, local variables cannot be declared to be `private` or `public`. The `private` specifier means available only to code within the class. Local variables are even more restricted, as other methods within the class cannot access them.

4.4 Method Parameters

A parameter to a method is a special local variable of that method. Whereas other local variables are assigned values within the method itself, parameters are assigned from outside the method during the method invocation. Consider the following interactive session using `CircleCalculator` (Figure 4.1):

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> c = new CircleCalculator();
> c.circumference(10)
62.8318
> x = 5
5
> c.circumference(x)
31.4159
```

In the first use of `circumference()`, the value 10 is assigned to the parameter `radius`, and the code within `circumference()` is then executed. In the second use of `circumference()`, the value of the variable `x` is assigned to `radius`, and then the code within `circumference()` is executed. In these examples we more precisely refer to 10 and `x` as *actual parameters*. These are the pieces of information *actually* supplied by the client code to the method during the method's invocation. Inside the definition of `circumference()` within `CircleCalculator`, `radius` is known as the *formal parameter*. A formal parameter is the name used during the *formal* definition of its method.

Note in our interactive session above that the variable `x` that is used as an actual parameter to the call of `circumference()`:

```
c.circumference(x)
```

has a different name than the actual parameter, `radius`. They are clearly two different variables. We could choose the names to be the same, as in:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> c = new CircleCalculator();
> radius = 5
5
> c.circumference(radius)
31.4159
```

but even in this case where the actual parameter has the same name as the formal parameter, we have two separate variables! Since the formal parameter `radius` is a special local variable, it has meaning only within the definition of `circumference()`. It cannot interfere with any variables anywhere else. The formal parameter exists in the computer's memory only while the method is being called. By contrast, the actual parameter exists before the call and after the call.

A formal parameter then is a placeholder so the code within the method can access a piece of data passed in by the client code. Like local variables, formal parameters are isolated from all other variables within the code or interactive session.

4.5 Summary

- A class defines the structure of an object. It is like a plan or blueprint that defines objects.
- Class names follow the identifier naming rules.
- Objects provide a service to clients.
- A client is the user of an object.
- Objects must be created to provide a service to clients.
- The `new` operator creates an object from its class description.
- An object is an active agent that provides a service to clients.
- Objects can possess both value and behavior.
- Object values are called attributes.
- Object behaviors are specified via methods.
- The dot (`.`) operator associates a method call with a particular object.
- Like a mathematical function, a method can accept information via parameters and compute a result.
- The types of a method's parameters must be specified within its parentheses.
- A method's result is indicated by a `return` statement.
- `public` class elements are accessible anywhere; `private` class elements are only accessible within the class itself.
- Comments come in three varieties: single line, multiple line, and documentation comments.
- Judicious comments improve the readability of source code.
- Variables declared within a method are called local variables.
- Local variables are isolated from all other variables within the code or interactive session.
- Parameters are special local variables used to communicate information into methods during their execution.
- A parameter used in the method definition is called a formal parameter.
- A parameter used during a method invocation is called an actual parameter.
- A method has exactly one definition, but it can be invoked zero or more times.

4.6 Exercises

1. What is a Java class?
2. What is an object? How is it related to a class?
3. How is object data specified?
4. How is object behavior specified?
5. How is a `public` class different from a `non-public` class?
6. What is a method?
7. What symbols delimit a class's body?
8. What do the `public` and `private` specifiers indicate within a class body?
9. What goes in the parentheses of a method's declaration?
10. What symbols delimit a method's body?
11. Within DrJava, how is a class compiled into bytecode?
12. What is client code?
13. What is a field?
14. What is an instance variable?
15. What is an attribute?
16. What is an operation?
17. How does client code create an instance of a class? Provide a simple example.
18. What happens when you try to evaluate an object directly within the interpreter? Loosely speaking, what the result mean?
19. What happens when clients attempt to evaluate the `private` data of an object?
20. Devise a class named `EllipseCalculator` that provides methods to compute the area and approximate perimeter of an ellipse given its major and minor radii (do a web search to see what is meant by these terms and to find appropriate formulas if you are uncertain). (Hint: to find the square root of a value in Java use `Math.sqrt()`, as in
$$\sqrt{x} = \text{Math.sqrt}(x)$$
21. What are the three kinds of comments supported by Java?
22. What is the purpose of comments?
23. What does the compiler do with the contents of comments?
24. Experiment putting comments inside of other comments. Under what circumstances is this possible?
25. How is a local variable different from an instance variable?
26. Rank the following kinds of variables from least restrictive most restrictive: `public` instance variable, local variable, `private` instance variable.

27. What advantage does a local variable have over an instance variable?
28. What is the difference between a formal parameter and an actual parameter?
29. What are the formal parameters of the `area` method of `RectangleCalculator` (Figure 4.2)?
30. How many different definitions may a method have?
31. How many times may a method be called?

Chapter 5

Boolean Expressions and Conditional Execution

Arithmetic expressions evaluate to numeric values; a Boolean expression, sometimes called a *predicate*, evaluates to `false` or `true`. Boolean expressions are essential for building conditional and iterative statements. The simplest Boolean expressions are `true` and `false`. More interesting Boolean expressions can be built by comparing program elements (variables and literals), checking for equality or inequality.

5.1 Relational Operators

Relational operators are used to compare two expressions. They evaluate to boolean values, `true` or `false`. Table 5.1 lists the relational operators available in Java.

Operator	Meaning
<code>==</code>	is equal to
<code>>=</code>	is greater than or equal to
<code><=</code>	is less than or equal to
<code><</code>	is less than
<code>></code>	is greater than
<code>!=</code>	is not equal to

Table 5.1: Relational operators

Boolean expressions can be entered into the interpreter:

Interactions

```

Welcome to DrJava.  Working directory is /Users/rick/java
> int x = 10;
> 10 < 20
true
> x < 20
true

```

```

> 10 > 20
false
> x > 20
false
> x == 10
true
> x == 20
false
> x != 10
false
> x != 20
true
> x <= 10
true
> x >= 10
true
> x <= 11
true
> x <= 9
false

```

(Be careful not to confuse the `>` Interactions pane prompt on the left of each expression with the `>` greater than operator that may be used in an expression.)

An expression like `10 < 20` is legal but of little use, since the expression `true` is equivalent, simpler, and less likely to confuse human readers. Boolean expressions are extremely useful when their truth values depend on the state of the program (for example, the value of a variable).

The relational operators are binary operators, and all have a lower precedence than any of the arithmetic operators; therefore, the expression

$$x + 2 < y / 10$$

is evaluated as if parentheses were placed as so:

$$(x + 2) < (y / 10)$$

Simple Boolean expressions, each involving one relational operator, can be combined into more complex Boolean expressions using the logical operators `&&` (and), `||` (or), and `!` (not). A combination of two or more Boolean expressions is called a *compound Boolean expression*. Table 5.2 shows how these logical operators work. Both `&&` and `||` are binary operators; that is, they require two operands, both of which must be Boolean expressions. Logical *not* (`!`) is a unary operator; it requires a single Boolean operand immediately to its right. Note that `||` is an *inclusive or*; that is, if either one or both of its operands is true, then the `||` expression is true.

Suppose you want to determine if an integer variable `x` is in the range `1...10`. In order for `x` to be within this range the following two conditions simultaneously must be true: `x >= 1` and `x <= 10`. The only numbers that meet both of these criteria are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

```

Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> x = 5;

```

e_1	e_2	$e_1 \ \&\& \ e_2$ (and)	$e_1 \ \ e_2$ (or)	$! \ e_1$ (not)
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 5.2: Logical operators. e_1 and e_2 and logical expressions.

```

> x >= 1 && x <= 10
true
> x = 15;
> x >= 1 && x <= 10
false
> x = 0;
> x >= 1 && x <= 10
false

```

$!$ has higher precedence than any binary operator. $\&\&$ has higher precedence than $||$. $\&\&$ and $||$ have lower precedence than any other binary operator except assignment. This means the expression

$$x \leq y \ \&\& \ x \leq z$$

is evaluated as if parentheses were added as shown here:

$$(x \leq y) \ \&\& \ (x \leq z)$$

Some programmers prefer to use the parentheses as shown here even though they are not required. The parentheses improve the readability of complex expressions, and the compiled code is no less efficient.

Suppose we want to see if a variable, x , is less than two other variables, y and z . Informally we may think, “Is x less than both y and z ?” and express this thought as:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> int x = 10, y = 20, z = 30;
> x < y && z
Error: And expression type

```

Logical **and** ($\&\&$) has lower precedence than less than ($<$), so the expression is evaluated as

$$(x < y) \ \&\& \ z$$

Recall the operands of $\&\&$ must be boolean expressions, but here z is an integer. Regrouping the parentheses would not help:

$$x < (y \ \&\& \ z)$$

Now neither operands of the `&&` are Boolean expressions! The correct way to express what we intend is:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 10, y = 20, z = 30;
> (x < y) && (x < z)
true
```

Thus the correct way to think about the comparison is to treat it as two independent clauses: “Is `x` less than `y`, and is `x` less than `z`?”

5.2 The if Statement

A program that behaves exactly the same way every time it is run is not very interesting. A chess program that makes the same moves every game it plays soon would be defeated easily. Useful software alters its behavior under varying environmental conditions, such as user input.

In Java, the fundamental statement for varying program behavior is the conditional statement. The simplest conditional statement uses `if` to optionally execute code. The simple `if` statement has the following general form:

```
if ( condition )
    body
```

where

- The reserved word `if` identifies an `if` statement.
- *condition* is a Boolean expression that determines whether or not the body will be executed. The condition must be enclosed within parentheses.
- *body* is either a single Java statement or a block of statements (called a *compound statement*) enclosed within curly braces. The curly braces are optional if the body consists of a single statement. Many developers always use curly braces as a matter of style, even when they are not required. If the body consists of only one statement and curly braces are not used, then the semicolon that terminates the statement in the body also terminates the `if` statement. If curly braces are used to delimit the body, a semicolon is not required after the close curly brace.

Figure 5.1 shows how program execution flows through a simple `if` statement.

The following interactive session illustrates the use of the simple `if` statement:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 20, y = 40;
> y
40
> if (x > 10)
    y = 50;
> y
```

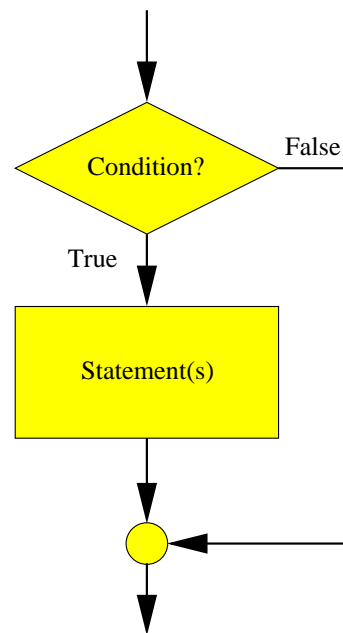


Figure 5.1: Execution flow in an if statement

```

50
> if (x > 100)
    y = 2;
> y
50

```

Notice that `y` is initially 40, as expected. The first `if` statement reassigns `y` to be 50, but the second `if` statement does not reassign `y` to 2. In the first case the Boolean expression `x > 10` is true, but in the second case `x > 100` is false. In both cases the reassignment of `y` is conditional on the value of `x`.

As shown above, a single `if` statement spans two lines. This is required neither in the DrJava interpreter nor by the Java language, but when we write Java programs it will improve the readability of the code. Also, we indented the body of the `if` on the second line of the statement. Again, neither the interpreter nor the Java language dictates this, but indentation is a standard convention to make the code more readable to humans. The compiler and interpreter are just as satisfied with

```

> if (x < 100) y = 2;
> y
2

```

Interactions

but to the human reader the indented body indicates that the code is optionally executed.

If more than one statement is to be included in the body, curly braces are required, as in

```

Welcome to DrJava. Working directory is /Users/rick/java
> int x = 20, y = 40, z = 60;
> x
20

```

Interactions

```

> y
40
> z
60
> if (x == 20) {
    x = 10;
    y = 5;
    z = 2;
}
> x
10
> y
5
> z
2

```

If the curly braces are omitted, the first statement is considered the body and the other statements are not part of the `if` statement. Remember, the indentation is for the convenience of the human reader not the compiler or interpreter. It is a good habit to enclose the body within curly braces even when it consists of a single statement. It is easy to introduce a logic error into a program if additional statements are added to the body later and the programmer forgets to add the then required curly braces.

Consider the following `if` statement that attempts to optionally assign `y`:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> int x = 20, y = 40;
> if (x < 10);
>     y = 2;
> y
2

```

It is important *not* to put a semicolon at the end of the line with the `if`. Here the semicolon terminates the `if` statement, but the indentation implies that the second line is intended to be part of the `if` statement. The compiler interprets the badly formatted `if` statement as if it were written as

Interactions

```

> if (x < 10)
>     ;
> x = 2;

```

The semicolon by itself represents an *empty statement* which is legal in Java and sometimes even useful. Here it means the `if` statement has an empty body. The assignment

```
y = 2;
```

is always executed since it is not part of the `if` statement body. The assignment statement is simply an independent statement following the `if` statement.

`TimeConverter2` (▣5.1) is a variation of `TimeConverter` (▣4.3) that returns a string in digital timer format (as in `03:41:07`):

```

public class TimeConverter2 {
    public String convert(int seconds) {
        // First compute hours, minutes, and seconds
        int hours = seconds/3600;
        seconds = seconds % 3600;
        int minutes = seconds/60;
        seconds = seconds % 60;
        // Next, format the output as 00:00:00
        String result = "";
        if (hours < 10) { // Print leading zero, if necessary
            result = result + "0";
        }
        result = result + hours + ":";
        if (minutes < 10) { // Print leading zero, if necessary
            result = result + "0";
        }
        result = result + minutes + ":";
        if (seconds < 10) { // Print leading zero, if necessary
            result = result + "0";
        }
        result = result + seconds;
        return result;
    }
}

```

Listing 5.1: TimeConverter2—converts seconds to hours:minutes:seconds

In TimeConverter2 (Figure 5.1) the `convert()` method works as follows:

- The first part that computes the values of the `hours`, `minutes`, and `seconds` variables remains the same as TimeConverter (Figure 4.3).
- As before, the second part assembles a string to return using the values computed from the first part. In this version, however, the format is as displayed on a digital timer: 00:00:00. It is easy to build a string consisting hours, minutes, and seconds separated by colons (:)—the string concatenation operator (+) works nicely. The more interesting problem is how to express 2 hours, 47 minutes, and 5 seconds as 02:47:05, instead of 2:47:5. We need to add an extra zero in front of numbers that are less than ten. For the case of hours:

```

if (hours < 10) { // Print leading zero, if necessary
    result = result + "0";
}
result = result + hours + ":";

```

we optionally concatenate a zero on the front of hours value. The same concept is applied to minutes and seconds.

As shown in this Interactions session, the method works as desired:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> t = new TimeConverter();
> t.convert(10000)
"02:46:40"
> t.convert(10025)
"02:47:05"
> t.convert(0)
"00:00:00"
> t.convert(1)
"00:00:01"
> t.convert(60)
"00:01:00"
```

5.3 The if/else Statement

The `if` statement allows the optional execution of a section of code. Either the code is executed, or it is not. Another common situation is the need to execute one section of code *or* another; that is, do *this* code *here* or do *that* code *there* depending on a given condition.

The `if` statement has an optional `else` clause. The general form of an `if/else` statement is

```
if ( condition )
    if body
else
    else body
```

Figure 5.2 shows the `if/else` flowchart.

The `else` clause contains an alternate body that is executed if the condition is false. Consider the following sequence of interactions:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 20, y = 40, z = 60;
> if (x == 20) {
    y = z;
} else {
    z = y;
}
> y
60
> z
60
> y = 40;
> if (x != 20) {
```

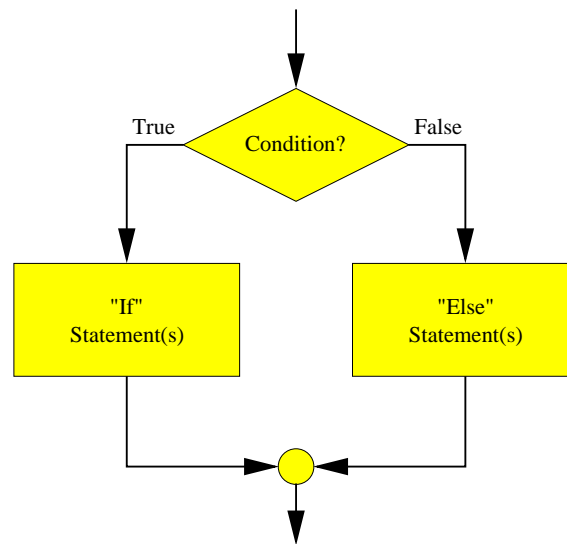


Figure 5.2: Execution flow in an if/else statement

```

    y = z;
  } else {
    z = y;
  }
> y
40
> z
40
>

```

The first if/else statement assigns z's value to y, while the second one assigns y's value to z.

The plain if statement is really a simplified form of the if/else statement. The plain if statement

```

if (x < 4) {
    y = 0;
}

```

could be written as

```

if (x < 4) {
    y = 0;
} else {}

```

The plain if form is simpler, though, and empty else bodies as shown here should not be used.

5.4 Summary

- The relational operators (==, !=, <, >, <=, and >=) evaluate to Boolean values.
- ! is the unary *not* operator.

- Boolean expressions can be combined via `&&` and `||`.
- The `if` statement can be used to optionally execute statements.
- A block groups a series of statements within a pair of curly braces (`{ }`).
- The `if` statement has an optional `else` clause to require the selection between two alternate paths of execution.
- The `if/else` statements can be nested to achieve arbitrary complexity.

5.5 Exercises

1. What is a predicate?
2. Can the two symbols that comprise the \leq operator in Java (`<=`) be written reversed (`=<`) and still represent \leq ?
3. Can a space appear between the two symbols that comprise Java's \leq operator (as in `< =`)?
4. Given that integer variables `x`, `y`, and `z` have the values 1, 2, and 3 respectively, determine the value of each of the following expressions:

a. <code>x == 2</code>	b. <code>x != 2</code>	c. <code>x <= 2</code>	d. <code>x >= 2</code>	e.
f. <code>x < 2</code>	g. <code>x > 2</code>	h. <code>x < 2 x == 2</code>	i. <code>!(x < y)</code>	j. <code>!(x <= y)</code>
k.	l.	m.	n.	o.
p.	q.	r.	s.	t.
5. Rewrite the following Boolean expressions in simpler form:

a. <code>!(x == 2)</code>	b. <code>x < 2 x == 2</code>	c. <code>!(x < y)</code>	d. <code>!(x <= y)</code>	e.
f.	g.	h.	i.	j.
k.	l.	m.	n.	o.
6. What Java conditional expression correctly represents the mathematical expression $10 \leq x \leq 100$?
7. Rewrite the following code fragment to achieve the same behavior without using an `else`:

```

if (x == y) {
    z = 0;
} else {
    z = 1;
}

```

8. The following code fragment works but is awkward. Rewrite it to achieve the same behavior without using the empty `if` body:

```

if (x == y) {}
else {
    z = 0;
}

```

- 9.

Chapter 6

More Complex Conditionals

Despite the initial simplicity of `if/else` statements as introduced in Chapter 5, things can get interesting when `if/else` statements are composed of other `if/else` statements. The result is that more exciting programs can be written.

6.1 Nested Conditionals

The statements in the body of the `if` or the `else` may be any Java statements, including other `if` statements. These nested `if` statements can be used to develop arbitrarily complex control flow logic. Consider `CheckRange` (Figure 6.1), whose `check()` method determines if a number is between 0...10, inclusive.

```
public class CheckRange {
    public boolean check(int value) {
        if ( value >= 0 ) { // First check
            if ( value <= 10 ) { // Second check
                return true;
            }
        }
        return false;
    }
}
```

Listing 6.1: `CheckRange`—Checks to see if a value is in the range 0–10

`CheckRange`'s `check()` method behaves as follows:

- The first condition is checked. If `value` is less than zero, the second condition is not evaluated and the statement following the outer `if` is executed. The statement after the outer `if` is the return of `false`.
- If the first condition finds `value` to be greater than or equal to zero, the second condition is then checked. If the second condition is met, `true` is returned.

Both conditions of this nested `if` must be met for the message to be printed. This program can be rewritten to behave the same way with only *one* `if` statement, as `NewCheckRange` (Figure 6.2) shows.

```
public class NewCheckRange {
    public boolean check(int value) {
        if ( value >= 0 && value <= 10 ) { // Compound check
            return true;
        }
        return false;
    }
}
```

Listing 6.2: `NewCheckRange`—Checks to see if a value is in the range 0–10

`NewCheckRange` uses a logical *and* to check both conditions at the same time. Its logic is simpler, using only one `if` statement, at the expense of a slightly more complex Boolean expression in its condition. The second version is preferable here, but a slight variation of `CheckRange` would be impossible to convert so only one `if` is used.

```
public class EnhancedRangeCheck {
    public String check(int value) {
        String result;
        if ( value >= 0 ) { // First check
            if ( value <= 10 ) { // Second check
                result = value + " is acceptable";
            } else {
                result = value + " is too large";
            }
        } else {
            result = value + " is too small";
        }
        return result;
    }
}
```

Listing 6.3: `EnhancedRangeCheck`—Accept only values within a specified range and provides more comprehensive messages

`EnhancedRangeCheck` (Figure 6.3) provides more specific messages instead of a simple acceptance or rejection. Exactly one of three messages is returned based on the method's parameter. A single `if` or `if/else` statement cannot choose from among more than two different execution paths.

`BinaryConversion` (Figure 6.4) uses a series of `if` statements to build a 10-bit binary string representing the binary equivalent of a decimal integer supplied by the user.

```
public class BinaryConversion {
```

```
public String decimalToBinary(int value) {  
    // Integer must be less than 1024  
    String result = ""; // Result string initially empty  
    if (value >= 0 && value < 1024) {  
        if (value >= 512) {  
            result += "1";  
            value %= 512;  
        } else {  
            result += "0";  
        }  
        if (value >= 256) {  
            result += "1";  
            value %= 256;  
        } else {  
            result += "0";  
        }  
        if (value >= 128) {  
            result += "1";  
            value %= 128;  
        } else {  
            result += "0";  
        }  
        if (value >= 64) {  
            result += "1";  
            value %= 64;  
        } else {  
            result += "0";  
        }  
        if (value >= 32) {  
            result += "1";  
            value %= 32;  
        } else {  
            result += "0";  
        }  
        if (value >= 16) {  
            result += "1";  
            value %= 16;  
        } else {  
            result += "0";  
        }  
        if (value >= 8) {  
            result += "1";  
            value %= 8;  
        } else {  
            result += "0";  
        }  
        if (value >= 4) {  
            result += "1";  
            value %= 4;  
        } else {
```

```

        result += "0";
    }
    if (value >= 2) {
        result += "1";
        value %= 2;
    } else {
        result += "0";
    }
    result += value;
}
return result;
}
}

```

Listing 6.4: BinaryConversion—Builds the 10-bit binary equivalent of an integer supplied by the user

In BinaryConversion:

- The outer `if` checks to see if the value provided is in the proper range. The program only works for nonnegative values, so the range is 0–1023.
- Each of inner `ifs` compare the entered integer against decreasing powers of two. If the number is large enough:
 - a 1 is appended to the string, and
 - that power of two’s contribution to the value is removed via the remainder operator.
- For the ones place, the remaining value can be appended without first checking it—value will be either 0 or 1 at that point.

`SimplerBinaryConversion` (Figure 6.5) simplifies `BinaryConversion`’s logic using only one `if` statement.

```

public class SimplerBinaryConversion {
    public String decimalToBinary(int value) {
        // Integer must be less than 1024
        String result = ""; // Result string initially empty
        if (value >= 0 && value < 1024) {
            result += value/512;
            value %= 512;
            result += value/256;
            value %= 256;
            result += value/128;
            value %= 128;
            result += value/64;
            value %= 64;
            result += value/32;
            value %= 32;
            result += value/16;
            value %= 16;

```

```

        result += value/8;
        value %= 8;
        result += value/4;
        value %= 4;
        result += value/2;
        value %= 2;
        result += value;
    }
    return result;
}
}

```

Listing 6.5: `SimplerBinaryConversion`—Re-implements `BinaryConversion` with only one *if* statement

The sole *if* statement in `SimplerBinaryConversion` ensures that the user provides an integer in the proper range. The other *if* statements originally found in `BinaryConversion` are replaced by a clever sequence of integer arithmetic operations. The two programs—`BinaryConversion` and `SimplerBinaryConversion`—behave identically but `SimplerBinaryConversion`'s logic is simpler.

6.2 Multi-way *if/else* Statements

What if exactly one of several actions should be taken? Nested *if/else* statements are required, and the form of these nested *if/else* statements is shown in `DigitToWord` (Figure 6.6).

```

public class DigitToWord {
    public String convert(int value) {
        String result;
        if ( value < 0 ) {
            result = "Too small";
        } else {
            if ( value == 0 ) {
                result = "zero";
            } else {
                if ( value == 1 ) {
                    result = "one";
                } else {
                    if ( value == 2 ) {
                        result = "two";
                    } else {
                        if ( value == 3 ) {
                            result = "three";
                        } else {
                            if ( value == 4 ) {
                                result = "four";
                            } else {
                                if ( value == 5 ) {

```

```

        result = "five";
    } else {
        result = "Too large";
    }
}
}
}
}
}
}
return result;
}
}

```

Listing 6.6: DigitToWord—Multi-way if formatted to emphasize its logic

Observe the following about DigitToWord's convert() method:

- It returns exactly one of eight strings depending on the user's input.
- Notice that each if body contains a single assignment statement and each else body, except the last one, contains an if statement. The control logic forces the program execution to check each condition in turn. The first condition that matches wins, and its corresponding if body will be executed. If none of the conditions are true, the last else's "too large" string will be returned.
- None of the curly braces used to delimit the if and else bodies are required since each body contains only a single statement (although a single deeply nested if/else statement is a mighty big statement).

DigitToWord is formatted according to the conventions used in earlier examples. As a consequence the mass of text drifts to the right as more conditions are checked. A commonly used alternative style, shown in RestyledDigitToWord (Figure 6.7), avoids this rightward drift.

```

public class RestyledDigitToWord {
    public String convert(int value) {
        String result;
        if ( value < 0 ) {
            result = "Too small";
        } else if ( value == 0 ) {
            result = "zero";
        } else if ( value == 1 ) {
            result = "one";
        } else if ( value == 2 ) {
            result = "two";
        } else if ( value == 3 ) {
            result = "three";
        } else if ( value == 4 ) {
            result = "four";
        } else if ( value == 5 ) {

```

```

        result = "five";
    } else {
        result = "Too large";
    }
    return result;
}
}

```

Listing 6.7: RestyledDigitToWord—Reformatted multi-way if

The formatting of `RestyledDigitToWord` does indeed hide the true structure of its logic, but it is so commonly used that it is regarded as acceptable by most programmers. The sequence of `else if` lines all indented to the same level identifies this construct as a multi-way if/else statement.

`DateTransformer` (¶6.8) uses a multi-way if/else to transform a numeric date in month/day (United States) format to an expanded English form, as in 2/14 → February 14. It also transforms the international day/month form to *Español*, as in 14-2 → 14 *febrero*.

```

public class DateTransformer {
    public String toEnglish(int month, int day) {
        String result;
        // Translate month
        if (month == 1) {
            result = "January ";
        } else if (month == 2) {
            result = "February ";
        } else if (month == 3) {
            result = "March ";
        } else if (month == 4) {
            result = "April ";
        } else if (month == 5) {
            result = "May ";
        } else if (month == 6) {
            result = "June ";
        } else if (month == 7) {
            result = "July ";
        } else if (month == 8) {
            result = "August ";
        } else if (month == 9) {
            result = "September ";
        } else if (month == 10) {
            result = "October ";
        } else if (month == 11) {
            result = "November ";
        } else {
            result = "December ";
        }
        // Add the day
    }
}

```

```
        result += day;

        return result;
    }

    public String toSpanish(int day, int month) {
        String result = day + " ";
        // Translate month
        if (month == 1) {
            result += "enero";
        } else if (month == 2) {
            result += "febrero";
        } else if (month == 3) {
            result += "marzo";
        } else if (month == 4) {
            result += "abril";
        } else if (month == 5) {
            result += "mayo";
        } else if (month == 6) {
            result += "junio";
        } else if (month == 7) {
            result += "julio";
        } else if (month == 8) {
            result += "agosto";
        } else if (month == 9) {
            result += "septiembre";
        } else if (month == 10) {
            result += "octubre";
        } else if (month == 11) {
            result += "noviembre";
        } else {
            result += "diciembre";
        }
        return result;
    }
}
```

Listing 6.8: DateTransformer—Transforms a numeric date into an expanded form

Valentine's Day and Christmas would be transformed as:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> DateTransformer dt = new DateTransformer();
> dt.toEnglish(2, 14)
"February 14"
> dt.toSpanish(14, 2)
"14 febrero"
> dt.toEnglish(12, 25)
```

```
"December 25"
> dt.toSpanish(25, 12)
"25 diciembre"
```

6.3 Errors in Conditional Expressions

Carefully consider each compound conditional used, such as

```
value >= 0 && value <= 10
```

found in `NewCheckRange` (Figure 6.2). Confusing logical *and* and logical *or* is a common programming error. The Boolean expression

```
value >= 0 || value <= 10
```

is known as a *tautology*. A tautology is a Boolean expression that is always true. What value could the variable `value` assume that would make this Boolean expression false? No matter its value, one or both of the subexpressions will be true, so the compound expression is always true. The *or* expression here is just a complicated way of expressing the value `true`.

Another common error is contriving compound Boolean expressions that are always false, known as *contradictions*. Suppose you wish to *exclude* values from a given range; that is, reject `0...10` and accept all other numbers. Is the Boolean expression in the following code fragment up to the task?

```
// All but 0, 1, 2, ..., 10
if ( value < 0 && value > 10 ) {
    /* Code to execute goes here . . . */
}
```

A closer look at the condition reveals it can *never* be true. What number can be both less than zero and greater than ten *at the same time*? None can, of course, so the expression is a contradiction and a complicated way of expressing false. To correct this code fragment, replace the `&&` operator with `||`.

6.4 Summary

- The relational operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`) evaluate to Boolean values.
- `!` is the unary *not* operator.
- Boolean expressions can be combined via `&&` and `||`.
- The `if` statement can be used to optionally execute statements.
- A block groups a series of statements within a pair of curly braces (`{}`).
- The `if` statement has an optional `else` clause to require the selection between two alternate paths of execution.
- The `if/else` statements can be nested to achieve arbitrary complexity.
- Complex Boolean expressions require special attention, as they are easy to get wrong.

6.5 Exercises

1. Rewrite the following Boolean expressions in simpler form:

- | | | | | |
|-------------------|------------------------|------------------|---------------------|----|
| a. $\neg(x == 2)$ | b. $x < 2 \vee x == 2$ | c. $\neg(x < y)$ | d. $\neg(x \leq y)$ | e. |
| f. | g. | h. | i. | j. |
| k. | l. | m. | n. | o. |

2. What is the simplest tautology?

3. What is the simplest contradiction?

Chapter 7

Modeling Real Objects

`CircleCalculator` (Figure 4.1) objects allow us to easily compute the circumference and area of a circle if we know its radius. We really need to create only one `CircleCalculator` object, because if we want to compute the circumference of a different circle, we only need to pass a different radius value to the `circumference()` method. Should we create more than one circle calculator object there would be no way to distinguish among them, except by their unique address identifiers; that is, given `CircleCalculator` (Figure 4.1):

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> cc1 = new CircleCalculator();
> cc2 = new CircleCalculator();
> cc1
CircleCalculator@2f4538
> cc2
CircleCalculator@a0ebc2
```

Otherwise, `cc1` and `cc2` behave identically. Similarly, the `convert()` method for `TimeConverter` objects computes the same result given the same input regardless of which `TimeConverter` object is used. We say that `CircleCalculator` objects and `TimeConverter` objects are *stateless*. A primitive-type variable like an integer has a state; its value defines its state. Given the following integer variables:

```
int x = 0, y = 4, z = 8;
```

each variable is in a different state, since each has a different value. The concept of state can be more complex than a simple value difference. We can define state however we choose; for example:

- `x`, `y`, and `z` are in different states, since they all have values different from each other.
- `y` and `z` are both in a nonzero state, while `x` is not.
- `x`, `y`, and `z` are all in the same state—nonnegative.

In this chapter we examine objects with state. Objects with state possess qualities that allow them to be distinguished from one another and behave differently over time.

7.1 Objects with State

Circle1 (Figure 7.1) is a slight textual variation of CircleCalculator (Figure 4.1) that has a profound impact how clients use circle objects. (We name it Circle1 since this is the first version of the circle class.)

```
public class Circle1 {
    private final double PI = 3.14159;
    private double radius;
    public Circle1(double r) {
        radius = r;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 7.1: Circle1—a slightly different circle class

In Circle1 (Figure 7.1) we add a new instance variable, `radius`, and new kind of element called a *constructor*:

- A new instance variable is introduced:

```
private double radius;
```

`radius` is really a variable because it lacks the `final` qualifier. Since it is private, however, clients cannot change its value because they cannot even see it.

- The method-looking construct named `Circle1` is called a *constructor*:

```
public Circle1(double r) {
    radius = r;
}
```

A constructor definition is similar to a method definition except:

- it has no return type specified,
- its name is the same as the class in which it is defined,
- it can be invoked only indirectly through the `new` operator.

Constructor code is executed when the `new` operator is used to create an instance of the class. Constructors allow objects to be properly initialized. Clients can pass to the constructor information that ensures the object begins its life in a well-defined state. In this case, the radius of the circle is set. Since `radius` is private, a client would have no other way to create a circle object and make sure that its radius was a certain value.

- Both of the original methods (`circumference()` and `area()`) now have empty parameter lists. Clients do not provide any information when they call these methods.

After saving and compiling `Circle1`, try out this session:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> Circle1 c1 = new Circle1(10.5),
      c2 = new Circle1(2.0);
> c1.circumference()
65.97339
> c2.circumference()
12.56636
```

Let us carefully examine this session:

- First we create two `Circle1` objects:

```
> Circle1 c1 = new Circle1(10.5),
      c2 = new Circle1(2.0);
```

The radius of each circle is specified within the parentheses. This `new` expression calls the constructor, and the argument provided by the client is passed to the constructor.

- We see that `c1` and `c2` have different circumferences.

```
> c1.circumference()
65.97339
> c2.circumference()
12.56636
```

We do not pass any information into the calls to `circumference()` because we do not need to. Each object maintains its own radius. It would be an error if we tried to pass a parameter to `circumference()` since `Circle1`'s `circumference()` method is not defined to accept any parameters. If you continue the interaction, you can demonstrate this:

```
Interactions
> c1.circumference(2.5)
Error: No 'circumference' method in 'Circle1' with arguments:
(double)
```

`Circle1` objects are *stateful*. There is a distinction between the objects referenced by `c1` and `c2`. Conceptually, one object is bigger than the other since `c1` has a larger radius than `c2`.¹ This is not the case with `CircleCalculator` objects. `CircleCalculator` objects are *stateless* because their `circumference()` methods always compute the same values given the same arguments. All `CircleCalculator` objects behave the same given the same inputs to their methods. Their variation in behavior was due to variation in clients' state, because the client could choose to pass different values as parameters.

A class may define more than one constructor. `Circle2` (Figure 7.2) includes two constructors and a new method:

¹This size distinction is strictly conceptual, however. Both `c1` and `c2` occupy the same amount of computer memory—they each hold two integer values, their own copies of `PI` and `radius`, and all integers require four bytes of memory.

```

public class Circle2 {
    private final double PI = 3.14159;
    private double radius;
    public Circle2(double r) {
        radius = r;
    }
    // No argument to the constructor makes a unit circle
    public Circle2() {
        radius = 1.0;
    }
    public double getRadius() {
        return radius;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}

```

Listing 7.2: Circle2—adds a new constructor to the Circle1 class

In Circle2 (Figure 7.2):

- The first constructor initializes the instance variable `radius` to the value passed in by the client at the time of creation.
- The second constructor makes a unit circle (`radius = 1`).
- The new method `getRadius()` returns the value of the instance variable `radius`. It is important to note that this does not allow clients to tamper with `radius`; the method returns only a copy of `radius`'s value. This means clients can see the value of `radius` but cannot touch it. We say that `radius` is a *read-only* attribute.

We can test the constructors in the Interactions pane:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> Circle2 circle = new Circle2(10.5);
> circle.getRadius()
10.5
> Circle2 circle2 = new Circle2();
> circle2.getRadius()
1.0

```

The constructors all have the same name (the same name as the class). They must be differentiated by different parameter lists, however. Here, one constructor accepts a single `double` value, while the other accepts no parameters. If no constructors were defined, the `radius` would default to zero, since numeric instance variables are automatically initialized to zero unless constructors direct otherwise.

When a class has more than one constructor, we say its constructor is *overloaded*. Java permits methods to be overloaded as well, but we have little need to do so and will not consider overloaded methods here. `Circle3` (Figure 7.3) gives a slightly different version of `Circle2`.

```
public class Circle3 {
    private final double PI = 3.14159;
    private double radius;
    public Circle3(double r) {
        radius = r;
    }
    public Circle3() {
        this(1.0); // Defer work to the other constructor
    }
    public double getRadius() {
        return radius;
    }
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 7.3: `Circle3`—one constructor calls another

One of the constructors in `Circle3` (Figure 7.3) contains the statement

```
this(1.0);
```

Here the reserved word `this` is used like a method call. It can be used in this way only within a constructor body. It allows one constructor to call another constructor within the same class. Here the parameterless constructor provides a default value for the “main” constructor that is responsible for initializing `radius`. We say the parameterless constructor defers the work to the other constructor, passing it the default argument. In this case, for `Circle2` both constructors initialize `radius` using an assignment statement. In `Circle3` only the first constructor uses assignment directly; the second constructor supplies a default value of 1.0 and invokes the services of the first constructor.

Constructors are optional. `CircleCalculator` (Figure 4.1) had no constructor. The compiler provides a *default constructor* for a class if the programmer omits a constructor definition for that class. The default constructor simply initializes all otherwise uninitialized numeric instance variables to zero. If a programmer defines at least one constructor for a class, the compiler will not create a default constructor for that class.

Constructors can do more than just initialize variables. They can do anything that methods can do.

7.2 Traffic Light Example

A traffic light is a real-world object that has a readily apparent state. A simple traffic light is shown in Figure 7.1. It can be red, yellow, green, or off (no lamps lit). Typically only one lamp is illuminated at one time. We wish to



Figure 7.1: A simple traffic light to be modeled in software

model such a traffic light in software. Our immediate purpose is to give us more experience defining classes and using objects, but such a software model could be put to good use in a program that a civil engineer might use to visualize a traffic flow simulation for an intersection or region of a city.

So far we have dealt with items of a mathematical nature. We have used numbers in computations, and the state of one of our circle objects can be represented by a number—its radius. Our traffic light model's behavior seems to have nothing to do with numbers or mathematics. It changes colors, and colors are not numbers. We do, however, have all the tools at our disposal to be able to model such an object.

First, we must describe exactly what one of our traffic light model objects is expected to do. We keep it simple here, but we will soon see that this simplicity will not necessarily hamper us when we want to make a more elaborate traffic light object. Our traffic light model must be able to:

- assume at the time of its creation a legitimate color, one of red, yellow, or green,
- change its color from its current color to the color that should follow its current color in the normal order of a traffic light's cycle, and
- indicate its current color.

For now, our tests will be text based, not graphical. A red traffic light will be printed as

```
[ (R) ( ) ( ) ]
```

A yellow light is rendered as

```
[ ( ) (Y) ( ) ]
```

and a green light is

```
[ ( ) ( ) (G) ]
```

(We consider a different text representation in § 10.2.)

While numbers may seem irrelevant to traffic light objects, in fact the clever use of an integer value can be used to represent a traffic light's color. Suppose we store an integer value in each traffic light object, and that integer represents the light's current color. We can encode the colors as follows:

- 1 represents red
- 2 represents yellow
- 3 represents green

Since remembering that 2 means “yellow” can be troublesome for most programmers, we define constants that allow symbolic names for colors to be used instead of the literal numbers. `RYGTrafficLight` (Figure 7.4) shows our first attempt at defining our traffic light objects:

```
// The "RYG" prefix stands for the Red, Yellow, and Green colors that
// this light can assume.
public class RYGTrafficLight {
    // Colors are encoded as integers
    private final int RED    = 1;
    private final int YELLOW = 2;
    private final int GREEN  = 3;

    // The current color of the traffic light
    private int color;

    // The constructor ensures that the light has a valid color
    // (starts out red)
    public RYGTrafficLight() {
        color = RED;
    }

    // Changes the light's color to the next legal color in its normal cycle.
    // The light's previous color is returned.
    // The pattern is:
    //      red --> green --> yellow
    //      ^           |
    //      |           |
    //      +-----+
    public void change() {
        if (color == RED) {
            color = GREEN; // Green follows red
        } else if (color == YELLOW) {
            color = RED; // Red follows yellow
        } else if (color == GREEN) {
            color = YELLOW; // Yellow follows green
        }
    }

    // Render the individual lamps
    public String drawLamps() {
        if (color == RED) {
            return "(R) ( ) ( )";
        } else if (color == GREEN) {
            return "( ) ( ) (G)";
        } else if (color == YELLOW) {

```



```

        return "( ) (Y) ( )";
    } else {
        return "**Error** "; // Defensive programming
    }
}
// Render the light in a crude visual way
public String draw() {
    return "[" + drawLamps() + "];"
}
}

```

Listing 7.4: RYGTrafficLight—simulates a simple traffic light

Some remarks about RYGTrafficLight (Figure 7.4):

- Comments are supplied throughout to explain better the purpose of the code.
- Color constants are provided to make the code easier to read.

```

private final int RED      = 1;
private final int YELLOW = 2;
private final int GREEN   = 3;

```

These constants are private because clients do not need to access them.

- The only way a client can influence a light's color is through `change()`. Note that the client cannot change a light's color to some arbitrary value like 34. The `change()` method ensures that light changes its color in the proper sequence for lights of this kind.
- The `draw()` and `drawLamps()` methods render the traffic light using text characters. The drawing changes depending on the light's current color. If because of a programming error within the RYGTrafficLight class the `color` variable is set outside of its legal range 0...3, the value returned is "[**Error**]". This is a form of *defensive programming*. While we expect and hope that everything works as it should, we make provision for a RYGTrafficLight object that is found in an illegal state. Should this string ever appear during our testing we will know our code contains a logic error and needs to be fixed. The `draw()` method renders the "frame" and uses the `drawLamps()` method to draw the individual lamps.

The following interactive sequence creates a traffic light object and exercises it through two complete cycles:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> t = new RYGTrafficLight();
> t.draw()
"[(R) ( ) ( )]"
> t.change(); t.draw()
"[( ) ( ) (G)]"
> t.change(); t.draw()
"[( ) (Y) ( )]"
> t.change(); t.draw()

```

```

"[(R) ( ) ( )]"
> t.change(); t.draw()
"[( ) ( ) (G)]"
> t.change(); t.draw()
"[( ) (Y) ( )]"
> t.change(); t.draw()
"[(R) ( ) ( )]"

```

Now we are in better position to see the value of the public and private access specifiers. Suppose the color instance variable were declared public. Clients then could create a traffic light object and set its color to, say, 34. The value 34 is not interpreted as one of the colors red, yellow, or green. By making color private, the RYGTrafficLight class developer can protect her traffic light objects from being abused either accidentally or maliciously. Clients cannot set a traffic light's color outside of its valid range. In fact, the client would need access to the source code for RYGTrafficLight (Figure 7.4) to even know that an integer was being used to control the light's color. (An alternate implementation might use three Boolean instance variables named red, yellow, and green, and a red light would have red = true, yellow = false, and green = false.)

7.3 RationalNumber Example

In mathematics, a *rational number* is defined as the ratio of two integers, where the second integer must be nonzero. Commonly called a *fraction*, a rational number's two integer components are called *numerator* and *denominator*. Rational numbers possess certain properties; for example, two fractions can have different numerators and denominators but still be considered equal (for example, $\frac{1}{2} = \frac{2}{4}$). Rational numbers may be added, multiplied, and reduced to simpler form. RationalNumber (Figure 7.5) is our first attempt at a rational number type:

```

public class RationalNumber {
    private int numerator;
    private int denominator;

    // num is the numerator of the new fraction
    // den is the denominator of the new fraction
    public RationalNumber(int num, int den) {
        if (den != 0) { // Legal fraction
            numerator = num;
            denominator = den;
        } else { // Undefined fraction changed to zero
            numerator = 0;
            denominator = 1;
        }
    }

    // Provide a human-readable string for the fraction, like "1/2"
    public String show() {
        return numerator + "/" + denominator;
    }

    // Compute the greatest common divisor for two integers m and n

```

```

// Uses Euclid's algorithm, circa 300 B.C.
// This method is meant for internal use only; hence, it is
// private.
private int gcd(int m, int n) {
    if (n == 0) {
        return m;
    } else {
        return gcd(n, m % n);
    }
}

// Returns a new RationalNumber consisting of the current
// fraction reduced to lowest terms
public RationalNumber reduce() {
    int factor = gcd(numerator, denominator);
    return new RationalNumber(numerator/factor,
                              denominator/factor);
}

// Returns a new RationalNumber consisting of the sum of this
// fraction and another fraction (other). The result is reduced
// to lowest terms.
public RationalNumber add(RationalNumber other) {
    int num1 = numerator * other.denominator,
        num2 = other.numerator * denominator;
    RationalNumber result = new RationalNumber(num1 + num2,
                                                denominator * other.denominator);
    return result.reduce();
}
}

```

Listing 7.5: RationalNumber—defines a rational number type

In RationalNumber (Listing 7.5):

- Two instance variables:

```

private int numerator;
private int denominator;

```

define the state of a rational number.

- The constructor ensures that the denominator of the new fraction will not be zero.

```

public RationalNumber(int num, int den) {
    if (den != 0) { . . .

```

Since the instance variables `numerator` and `denominator` are private and none of the methods change these variables, it is impossible to have an undefined rational number object.

- The `show()` method:

```
public String show() { . . .
```

allows a rational number to be displayed in a human-readable form, as in “1/2.”

- The `gcd()` method:

```
private int gcd(int m, int n) { . . .
```

computes the greatest common divisor (also called greatest common factor) of two integers. This method was devised by the ancient Greek mathematician Euclid around 300 B.C. This method is interesting because it calls itself. A method that calls itself is called a *recursive* method. All proper recursive methods must involve a conditional (such as an `if`) statement. The conditional execution ensures that the method will not call itself under certain conditions. Each call to itself should bring the method closer to the point of not calling itself; thus, eventually the recursion will stop. In an improperly written recursive method, each recursive call does not bring the recursion closer to its end, and infinite recursion results. Java’s runtime environment, catches such infinite recursion and generates a runtime error.

- The `reduce()` method:

```
public RationalNumber reduce() {
    int factor = gcd(numerator, denominator);
    return new RationalNumber(numerator/factor,
                              denominator/factor);
}
```

returns a new rational number. It uses the `gcd()` method to find the greatest common divisor of the numerator and denominator. Notice that a new rational number object is created and returned by the method.

- The `add()` method:

```
public RationalNumber add(RationalNumber other) {
```

accepts a reference to a rational number as a parameter and returns a new rational number as a result. It uses the results of the following algebraic simplification:

$$\frac{a}{b} + \frac{c}{d} = \left(\frac{d}{d}\right) \cdot \frac{a}{b} + \frac{c}{d} \cdot \left(\frac{b}{b}\right) = \frac{ad}{bd} + \frac{bc}{bd} = \frac{ad+bc}{bd}$$

The expression

```
new RationalNumber(num1 + num2, denominator * other.denominator)
```

creates a new `RationalNumber` object. We assign its value to a local `RationalNumber` variable named `result` and then return the reduced value of `result`. The call to `reduce()` creates a new rational number object from the rational number just created; thus, `add()` creates two rational number objects each time it is called, but it returns only one of them (the reduced form).

7.4 Object Aliasing

We have seen that Java provides some built-in primitive types such as `int`, `double`, and `boolean`. We also have seen how to define our own programmer-defined types with classes. When we declare a variable of a class type, we call the variable a *reference variable*.

Consider a very simple class definition, `SimpleObject` (Listing 7.6):

```
public class SimpleObject {
    public int value;
}
```

Listing 7.6: `SimpleObject`—a class representing very simple objects

The variable `value` is public, so it is fair game for any client to examine and modify. (`SimpleObject` objects would not be very useful within a real program.)

Object references and primitive type variables are fundamentally different. We do not use the `new` operator with primitive types, but we must create objects using `new`. The interpreter illustrates this difference:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> SimpleObject obj;
> obj.value = 2;
NullPointerException:
  at sun.reflect.UnsafeFieldAccessorImpl.ensureObj(Unsafe ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.getInt(Un ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.get(Unsaf ...
  at java.lang.reflect.Field.get(Field.java:357)
> obj.value
NullPointerException:
  at sun.reflect.UnsafeFieldAccessorImpl.ensureObj(Unsafe ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.getInt(Un ...
  at sun.reflect.UnsafeIntegerFieldAccessorImpl.get(Unsaf ...
  at java.lang.reflect.Field.get(Field.java:357)
> int number;
> number = 2;
> number
2
```

Simply declaring `obj` is not enough to actually use it, but simply declaring `number` is sufficient to allow it to be used in any legal way ints can be used.

For another example of the differences between primitive types and objects, consider the following interactive sequence:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> int i1 = 1, i2 = 2;
```

```
> i1
1
> i2
2
> i1 = i2;
> i1
2
> i2
2
> i1 = 0;
> i1
0
> i2
2
> SimpleObject o1 = new SimpleObject(),
                o2 = new SimpleObject();
> o1.value = 1;
> o1.value
1
> o2.value = 2;
> o2.value
2
> o1 = o2;
> o1.value
2
> o2.value
2
> o1.value = 0;
> o1.value
0
> o2.value
0
```

When one integer variable is assigned to another, as in

```
i1 = i2;
```

the value of the right-hand variable is assigned to the left-hand variable. `i1` takes on the value of `i2`. Reassigning `i1` does not effect `i2`; it merely gives `i1` a new value. `i1` and `i2` are distinct memory locations, so changing the value of one (that is, storing a new value in its memory location) does not change the value of the other. Figure 7.2 illustrates primitive type assignment.

A reference, on the other hand, is different. An object reference essentially stores the memory address of an object. A reference *is* a variable and is therefore stored in memory, but it holds a memory address, not a “normal” numeric data value.² In the statement

```
o1 = new SimpleObject();
```

²In fact, a reference *is* a number since each memory location has a unique numeric address. A reference is interpreted as an address, however, and the Java language does not permit arithmetic operations to be performed on references.

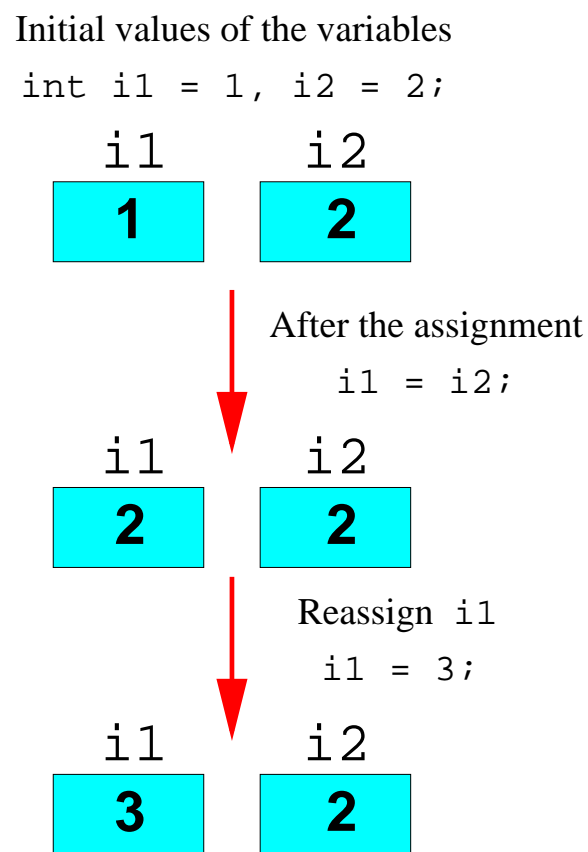


Figure 7.2: Assignment of Primitive Types

the `new` operator allocates memory for a new `SimpleObject` object and returns the address where the object was allocated. This address is then stored in `o1`. Interacting with the object, as in

```
o1.value = 2;
```

will affect the object at the address stored in `o1`. The statement

```
i1 = i2;
```

causes `i1` and `i2` to hold identical integer values. The statement

```
o1 = o2;
```

causes `t1` and `t2` to hold identical object *addresses*. If an object is modified via `o1`, then the object referenced via `o2` is also modified because `o1` and `o2` refer to exactly the same object! When two references refer to the same object we say that the references are *aliases* of each other. It is helpful to visualize object references as variables that point to the object they reference. Figure 7.3 shows object references as pointers (arrows).

Pictorially, `o1` is an alias of `o2` because both `o1` and `o2` point to the same object.

7.5 Summary

- Real-world objects have state; Java objects model state via instance variables.

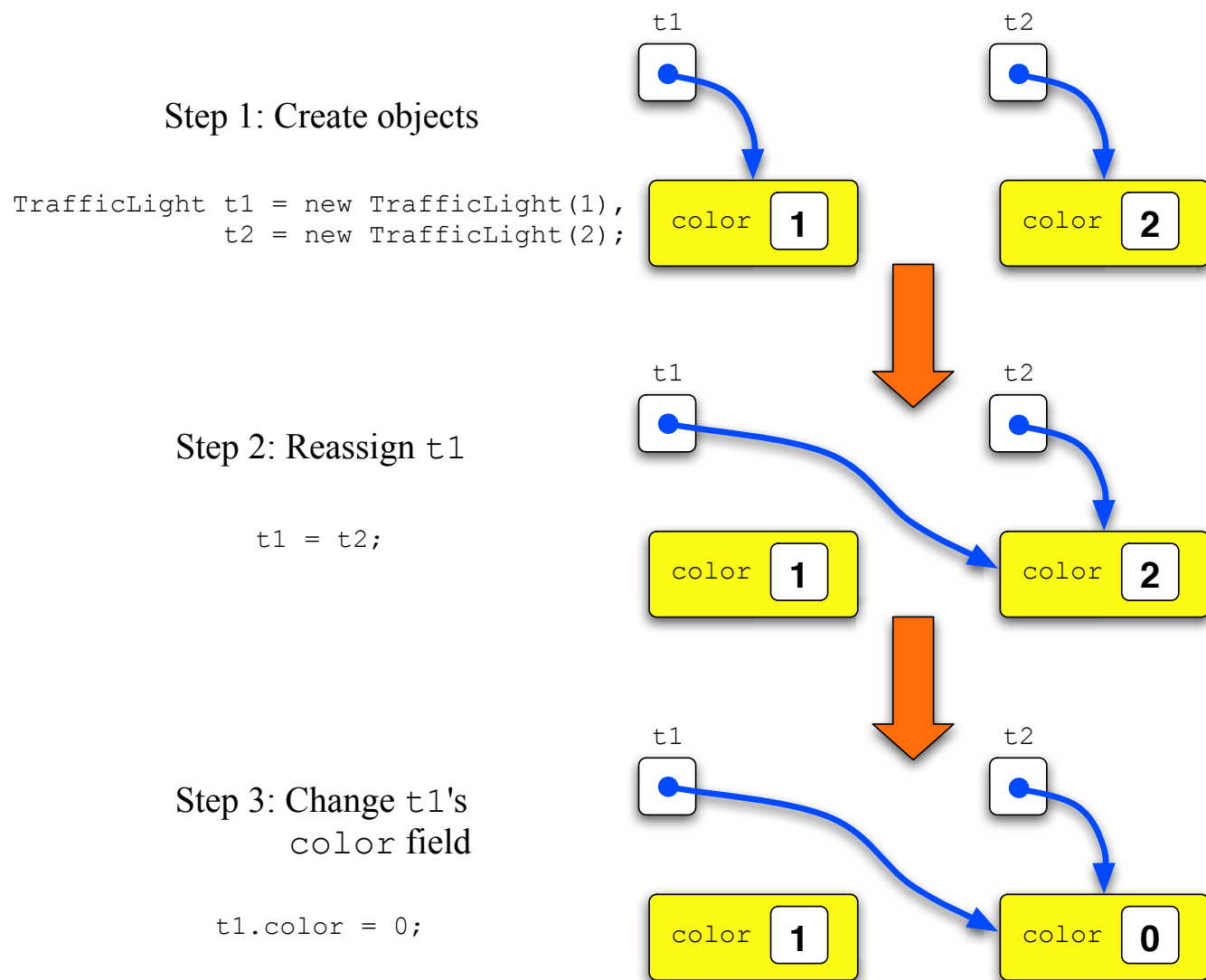


Figure 7.3: Assignment of Reference Types

- A class's constructor is called when an instance (object) is created via `new`.
- A constructor is similar to a method (but it technically is not a method), but it specifies no return type and is "called" indirectly only when `new` is used.
- A recursive method calls itself in its definition.
- When two object references refer to the same object we say one reference aliases the other reference.

7.6 Exercises

1. How does an object maintain its state?
2. In what three ways are a constructor different from a method?
3. What does it mean for a constructor to be overloaded?
4. How can one constructor directly call another constructor within the same class?

5. What does the compiler do if you do not provide a constructor for a class?
6. Is it legal for a method to specify an object reference as a parameter?
7. Is it legal for a method to return an object reference as a result?
8. What does it mean for a method to be recursive?
9. When speaking on objects, what is an alias?
10. Implement the alternate version of `RYGTrafficLight` mentioned in 7.2 that uses the three Boolean variables `red`, `yellow`, and `green` instead of the single integer `color` variable. Test it out to make sure it works.
11. Create a class named `Point` that is used to create (x,y) point objects. The coordinates should be `double` values. Provide a method that computes the distance between two points.

Chapter 8

Living with Java

This chapter deals with some miscellaneous issues: program output, formatting, errors, constants, and encapsulation.

8.1 Standard Output

The Java runtime environment (JRE) supplies a plethora of standard objects. The most widely used standard object is `System.out`. `System.out` is used to display information to the “console.” The console is an output window often called *standard out*. Under Unix/Linux and Mac OS X, standard out corresponds to a shell. Under Microsoft Windows, it is called the command shell (`cmd.exe`). In the DrJava environment, standard out is displayed in the Console pane which is made visible by selecting the **Console** tab.

The `println()` method from `System.out` outputs its argument to the console. If executed within the DrJava interpreter, it outputs to the Interactions pane as well.

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> System.out.println(10);
10
\end{verbatim}
A quick check of the Console pane reveals:
\begin{Console}
10
\end{Console}
Here is a little more elaborate session:
\begin{Interactions}
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 2, y = 3;
> System.out.println("The sum of " + x + " and " + y +
    " is " + (x + y));
The sum of 2 and 3 is 5
> System.out.println(x + " + " + y + " = " + (x + y));
2 + 3 = 5

```

The `System.out.print()` method works exactly like `println()`, except subsequent output will appear on the same line. Compare the following statements:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> System.out.println(10); System.out.println(20);
10
20
> System.out.print(10); System.out.print(20);
1020>
```

The `System.out.println()` and `System.out.print()` methods are overloaded since they will accept any primitive type or reference type.

Well-placed `println()` statements are invaluable for understanding what a program is doing when it is running.

8.2 Formatted Output

A companion method of `System.out.println()` is `System.out.printf()`. The `System.out.printf()` method provides formatted output for primitive types and strings. Special control codes denote display widths and decimal places. An example call is

```
System.out.printf("[%5d]-[%7.4f]: %s %n", 45, 3.14159, "Test values");
```

which outputs

```
[    45]-[ 3.1416]: Test values
```

The `printf()` method accepts a comma-separated list of arguments. The first argument, called the *format string*, is a string literal enclosed by quotation marks. The other arguments (if any) depend on formatting control codes within the format string. The contents of the format string are printed literally with the exception of the control codes which denote arguments that follow the format string. The first control code corresponds to the first argument after the format string, the second control code corresponds to the second argument after the format string, etc. Table 8.1 lists some commonly used control codes. The `printf()` method provides many other capabilities described in the class documentation. For example, if an optional zero appears before the width specifier in a numeric control code, as in

```
%04d
```

then the number is padded with leading zeros instead of leading blank spaces to make the number fill the specified width.

Notice in our example that the square brackets, dash, colon, and spaces appear as they do in the format string. The value 45 is displayed with three spaces preceding it to make a total of five spaces for the value. For 3.14159, a width of seven spaces is allotted for the entire value, and three decimal places are to be displayed. Note that the number of slots of characters between the displayed square brackets is seven, and the value is rounded to four decimal places. The string appears literally as it does within the quotes.

If the format string provides a control code like `%d` and `%f` that requires a trailing argument, the programmer must ensure that an argument of the appropriate type appears in its proper place in the list that follows the format

Control Code	Meaning
%d	Decimal (that is, base 10) integer (any integral type). An optional single number between the % and the d specifies field width for right justification.
%f	Floating point number (float or double). Optional field width and/or number of decimal places to display can appear between the % symbol and the f.
%s	String. Can be a string literal (as in this example) or a string variable.
%n	Platform-independent end-of-line character. The major operating systems (Unix/Linux, Microsoft Windows, and Macintosh) use different characters to signify a newline. This control code is translated into the proper newline character for the platform upon which the program is running.
%%	Percent symbol. Since % indicates that a control code symbol follows, this provides a way to print a literal percent symbol.

Table 8.1: Control codes used in the `System.out.printf()` format string

string. The compiler does *not* check for compliance. If too few trailing arguments are provided or the type of the argument does not agree with the control code, the program will generate a runtime error when the offending `printf()` statement is executed. Extra trailing arguments are ignored.

8.3 Keyboard Input

DrJava's Interactions pane provides a *read-evaluate-print* loop:

1. The user's keystrokes are read by the interpreter.
2. Upon hitting **Enter**, the user's input is evaluated, if possible.
3. The result of the evaluation, if any, is then displayed.

This way of user interaction is built into DrJava and we do not need to provide code to make it work. On the other hand, standalone programs are completely responsible for obtaining user input at the right time and providing the proper feedback to the user. This interaction must be handled by statements within the program.

We have seen how `System.out.print()`, `System.out.println()`, and `System.out.printf()` can be used to provide output. User input can be handled by `Scanner` objects. The `Scanner` class is found in the `java.util` package and must be imported for use within programs. `SimpleInput` (Figure 8.1) shows how a `Scanner` object can be used:

```
import java.util.Scanner;

public class SimpleInput {
    public void run() {
        int val1, val2;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter an integer: ");
```

```
    val1 = scan.nextInt();
    System.out.print("Please enter another integer: ");
    val2 = scan.nextInt();
    System.out.println(val1 + " + " + val2 + " = " + (val1 + val2));
}
}
```

Listing 8.1: SimpleInput—adds two integers

Compile this class and try it out:

Interactions

Welcome to DrJava. Working directory is /Users/rick/Documents/src/java/DrJava
> new SimpleInput().run();
Please enter an integer: 34
Please enter another integer: 10
34 + 10 = 44

The Interactions pane prints the prompt message (*Please enter an integer:*) without its familiar “>” prompt. After entering the values, their sum is then printed. Notice that you can enter both numbers together at the first prompt, separated by at least one space, and then press **Enter**. The first `nextInt()` grabs the first integer from the keyboard buffer, and the next call to `nextInt()` grabs the remaining value left over on the same line.

The `Scanner`’s constructor here accepts a standard `System.in` object which is associated with the keyboard. It can be associated with other input streams such as text files, but we will not consider those options here.

`Scanner` objects have a variety of methods to read the different kinds of values from the keyboard. Some of these methods are shown in Table 8.2:

Some Methods of the Scanner Class	
<code>int nextInt()</code>	Returns the integer value typed in by the user. This method produces a runtime error if the key sequence typed in cannot be interpreted as an <code>int</code> .
<code>double nextDouble()</code>	Returns the double-precision floating point value typed in by the user. This method produces a runtime error if the key sequence typed in cannot be interpreted as a <code>double</code> .
<code>String next()</code>	Returns the next string token typed in by the user. Tokens are separated by spaces.
<code>String nextLine()</code>	Returns the string typed in by the user including any spaces.

Table 8.2: A subset of the methods provided by the `Scanner` class

8.4 Source Code Formatting

Program comments are helpful to human readers but ignored by the compiler. The way the source code is formatted is also important to human readers but is of no consequence to the compiler. Consider

ReformattedCircleCalculator (Figure 8.2), which is a reformatted version of CircleCalculator (Figure 4.1):

```

public
class
ReformattedCircleCalculator{final
                                private double PI=3.14159
;public double circumference
                                (double
radius) {return 2*PI
                                *
radius;}public double
                                area(double
radius) {return PI*radius*radius;}}
```

Listing 8.2: ReformattedCircleCalculator—A less than desirable formatting of source code

To an experienced Java programmer CircleCalculator is easier to understand more quickly than ReformattedCircleCalculator. The elements of CircleCalculator are organized better. What are some distinguishing characteristics of CircleCalculator?

- Each statement appears on its own line. A statement is not unnecessarily split between two lines of text. Visually, one line of text implies one action (statement) to perform.
- Every close curly brace aligns vertically with the line that ends with the corresponding open curly brace. This makes it easier to determine if the curly braces match and nest properly. This becomes very important as more complex programs are tackled since curly braces frequently can become deeply nested.
- Contained elements like the circumference() method within the class and the statements within the circumference() method are indented several spaces. This visually emphasizes the fact that the elements are indeed logically enclosed.
- Spaces are used to spread out statements. Space around the assignment operator (=) and addition operator (+) makes it easier to visually separate the operands from the operators and comprehend the details of the expression.

Most software organizations adopt a set of *style guidelines*, sometimes called *code conventions*. These guidelines dictate where to indent and by how many spaces, where to place curly braces, how to assign names to identifiers, etc. Programmers working for the organization are required to follow these style guidelines for code they produce. This allows a member of the development team to more quickly read and understand code that someone else has written. This is necessary when code is reviewed for correctness or when code must be repaired or extended, and the original programmer is no longer with the development team.

One good set of style guidelines for Java can be found at Sun's web site:

<http://www.javasoft.com/codeconv/index.html>

The source code in this book largely complies with Sun's code conventions.

8.5 Errors

Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program. A programming error falls under one of three categories:

- compile-time error
- runtime error
- logic error

Compile-time errors. A compile-time error results from the misuse of the language. A *syntax error* is a common compile-time error. For example, in English one can say

The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the Java statement

```
x = 10 + 2;
```

is syntactically correct because it obeys the rules for the structure of an assignment statement described in § 2.2. However, consider replacing this assignment statement with a slightly modified version:

Interactions

```
> 10 + 2 = x;  
Error: Bad left expression in assignment
```

When the interpreter attempts to compile `BadAssignment`, it issues an error message.¹

Compilers have the reputation for generating cryptic error messages. They seem to provide little help as far as novice programmers are concerned. The message issued here is actually very useful. The “left expression” is the expression to the left of the assignment operator (`=`). It makes no sense to try to change the value of `10 + 2`—it is always 12.

`CircleCalculatorWithMissingBrace` (Figure 8.3) contains another kind of syntax error. All curly braces must be properly matched, but in `CircleCalculatorWithMissingBrace` the curly brace that closes the `circumference()` method is missing.

```
public class CircleCalculatorWithMissingBrace {  
    final double PI = 3.14159;
```

¹The messages listed here are generated by DrJava’s compiler. Other compilers should issue similar messages, but they may be worded differently.

```
double circumference(double radius) {  
    return 2 * PI * radius;  
double area(double radius) {  
    return PI * radius * radius;  
}  
}
```

Listing 8.3: CircleCalculatorWithMissingBrace—A missing curly brace

Figure 8.1 shows the result of compiling CircleCalculatorWithMissingBrace (Listing 8.3). As shown in Figure 8.1,

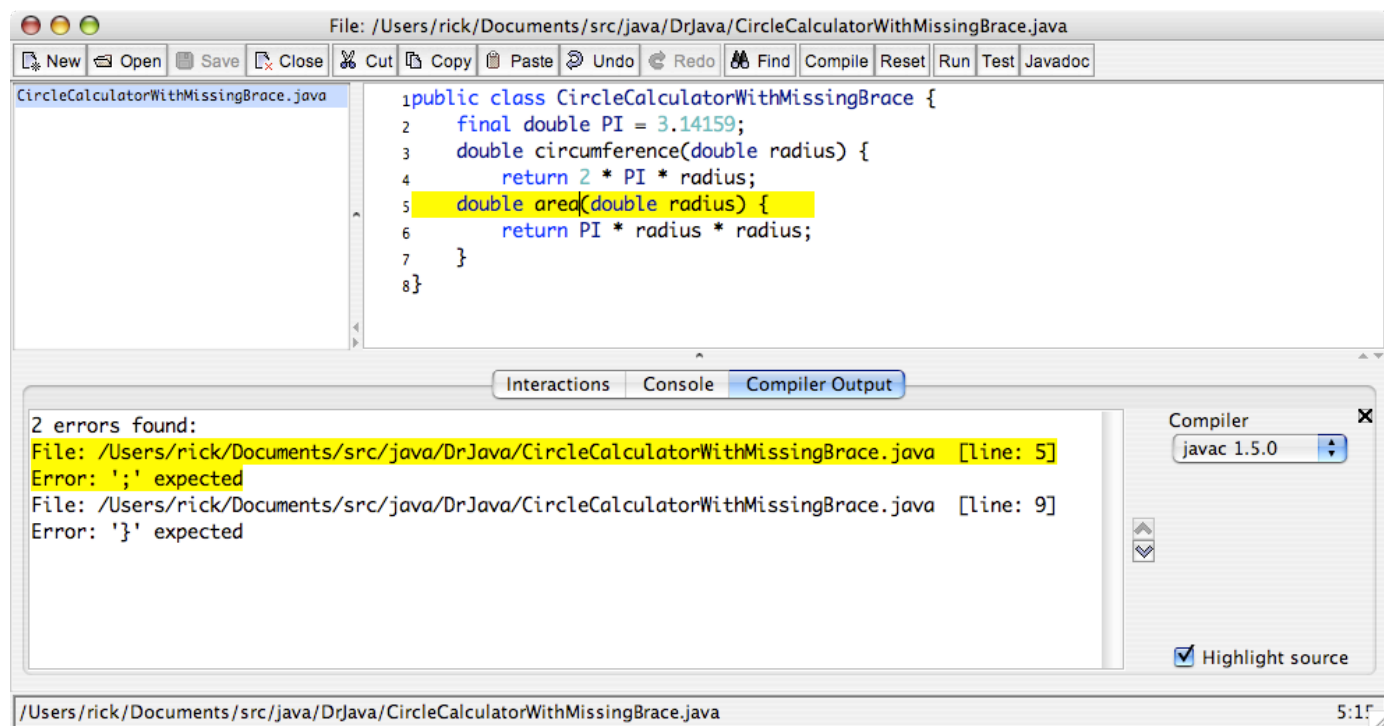


Figure 8.1: Result of compiling a Java source file with a compile-time error

an attempt to compile this code results in the following messages in the CompilerOutput pane:

Compiler Output

```
2 errors found:  
File:  
CircleCalculatorWithMissingBrace.java  
[line: 5]  
Error: ';' expected  
File:  
CircleCalculatorWithMissingBrace.java  
[line: 9]  
Error: '}' expected
```

This report is an example of how the compiler does indeed detect an error, but it does not appear to be the error that we expected! The first error it reports is on Line 5, which is in fact where the problem lies. The compiler

faithfully attempts to translate into bytecode the source code provided by the programmer, and it reports errors only when it encounters an illegal situation. In Java, it is illegal to define a method directly within another method. Said another way, Java does not permit nested method definitions. When the compiler is scanning Line 5, it assumes the programmer is attempting to declare a `double` variable named `area` within the `circumference()` method, which is legal to do. When it sees the left parenthesis, it then detects an attempt to define a method named `area` inside another method (`circumference()`), which is illegal. Had the programmer instead used a semicolon (`;`), the declaration would have been legal. The compiler then trudged on, eventually determining that the three opening braces (`{`) it counted were not properly matched up with an equal number of closing braces. (It ran out of source code on Line 9 while only finding two matching closing braces.)

`MissingDeclaration` (Figure 8.4) contains another error of omission. It is an error to use a variable that has not been declared.

```
class MissingDeclaration {  
    int attemptToUseUndeclaredVariable() {  
        x = 2;  
        return x;  
    }  
}
```

Listing 8.4: `MissingDeclaration`—A missing declaration

`MissingDeclaration` uses variable `x`, but `x` has not been declared. This type of compile-time error is not a syntax error, since all of the statements that are present are syntactically correct. The compiler reports

Compiler Output

```
2 errors found:  
File:  
MissingDeclaration.java  
[line: 3]  
Error: cannot find symbol  
symbol  : variable x  
location: class MissingDeclaration  
File:  
MissingDeclaration.java  
[line: 4]  
Error: cannot find symbol  
symbol  : variable x  
location: class MissingDeclaration  
s
```

Here, the compiler flags both lines that attempt to use the undeclared variable `x`. One “error,” forgetting to declare `x`, results in two error messages. From the programmer’s perspective only one error exists—the missing declaration. The compiler, however, cannot determine what the programmer intended; it simply notes the two uses of the variable that has never been declared, printing two separate error messages. It is not uncommon for beginning programmers to become disheartened when the compiler reports 50 errors and then be pleasantly surprised when one simple change to their code removes all 50 errors!

The message “cannot find symbol” means the compiler has no record of the symbol (variable *x*) ever being declared. The compiler needs to know the type of the variable so it can determine if it is being used properly.

Runtime errors. The compiler ensures that the structural rules of the Java language are not violated. It can detect, for example, the malformed assignment statement and the use of a variable before its declaration. Some violations of the language cannot be detected at compile time, however. Consider `PotentiallyDangerousDivision` (Figure 8.5):

```
class PotentiallyDangerousDivision {
    // Computes the quotient of two values entered by the user.
    int divide(int dividend, int divisor) {
        return dividend/divisor;
    }
}
```

Listing 8.5: `PotentiallyDangerousDivision`—Potentially dangerous division

The expression

`dividend/divisor`

is potentially dangerous. Since the two variables are of type `int`, integer division will be performed. This means that division will be performed as usual, but fractional results are dropped; thus, $15 \div 4 = 3$, not 3.75, and rounding is not performed. If decimal places are needed, floating point numbers are required (§ 2.1).

This program works fine (ignoring the peculiar behavior of integer division), until zero is entered as a divisor:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> d = new PotentiallyDangerousDivision();
> d.divide(15, 4)
3
> d.divide(15, 0)
ArithmeticException: / by zero
    at PotentiallyDangerousDivision.divide(PotentiallyDangerousDiv...
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcc...
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM...
    at java.lang.reflect.Method.invoke(Method.java:585)
```

Division by zero is undefined in mathematics and is regarded as an error in a Java program. This type of error, however, cannot be detected by the compiler since the value of the divisor is not known until the program is executing (runtime). It would be inappropriate (and certainly annoying) if the compiler issued a message in every instance where division by zero is a possibility. Fortunately, the Java runtime environment checks for various errors such as division by zero. When such an error is encountered, a message is displayed and the program’s execution is normally terminated. This action is called *throwing an exception*. The message indicates that the problem is division by zero and gives the source file and line number of the offending statement.

Logic errors. Consider the effects of replacing the statement

```
return dividend/divisor;
```

in `PotentiallyDangerousDivision` (Figure 8.5) with the statement:

```
return divisor/dividend;
```

The program compiles with no errors. It runs, and unless a value of zero is entered for the dividend, no runtime errors arise. However, the answer it computes is in general not correct. The only time the correct answer is printed is when `dividend = divisor`. The program contains an error, but neither the compiler nor the runtime environment is able to detect the problem. An error of this type is known as a *logic error*.

Beginning programmers tend to struggle early on with compile-time errors due to their unfamiliarity with the language. The compiler and its error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of syntax errors decrease and the number of logic errors increase. Unfortunately, both the compiler and runtime environments are powerless to provide any insight into the nature and/or location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers are frequently used to help locate and fix logic errors, but these tools are far from automatic in their operation.

The runtime exceptions mentioned in § 8.5 arise from logic errors. In the division by zero example, programmers can take steps to ensure such division by zero does not occur. `BetterDivision` (Figure 8.6) shows how it might be done.

```
class BetterDivision {
    // Computes the quotient of two values entered by the user.
    int divide(int dividend, int divisor) {
        if (divisor == 0) {
            System.out.println("Warning! Division by zero, result " +
                               "is invalid");
            return 2147483647; // Return closest int to infinity!
        } else {
            return dividend/divisor;
        }
    }
}
```

Listing 8.6: `BetterDivision`—works around the division by zero problem

`BetterDivision` (Figure 8.6) avoids the division by zero runtime error that causes the program to terminate prematurely, but it still alerts the user that there is a problem. Another application may handle the situation in a different way; for example, use some default value for `divisor` instead of zero.

Errors that escape compiler detection (runtime errors and logic errors) are commonly called *bugs*. Because of the inability of the compiler to detect these problems, such bugs are the major source of frustration for developers.

8.6 Constants Revisited

As mentioned earlier (§ 2.2), the proper use of constants can make Java code more readable. Constants have other advantages besides readability. Here we consider the advantages.

Why not use a literal value instead of a symbolic constant (for example, 3.14159 vs. `PI`)? A defined constant has several advantages over a literal value:

- The symbolic constant is potentially more readable. For example, it may be better to use the constant `MOLE` than to use its literal value `6.023e23` since the literal number may easily blend in with other literal values in an expression and the symbolic constant stands out with clear meaning.
- The symbolic constant hides specific details that are not relevant to the task at hand. For example, the area of a circle is $A = \pi r^2$. If the constant `PI` is defined somewhere, we can write the assignment

```
area = PI * radius * radius;
```

without worrying about how many decimal places of precision to use for `PI`.

- The specific information about the value of the constant is located in exactly one place. If the value must be changed, the programmer need only change one line in the program instead of possibly hundreds of lines throughout the program where the value is used. Consider a software system that must perform a large number of mathematical computations (any program that uses graphics extensively would qualify). Some commercial software systems contain several million lines of code. Calculations using the mathematical constant π could appear hundreds or thousands of times within the code. Suppose the program currently uses 3.14 for the approximation, but users of the program demand more precision, so the developers decide to use 3.14159 instead. Without constants, the programmer must find every occurrence of 3.14 in the code and change it to 3.14159. It is a common mistake to change most, but not all, values, thus resulting in a program that has errors. If a constant is used instead, like

```
final double PI = 3.14;
```

and the symbolic constant `PI` is used throughout the program where π is required, then only the constant initialization need be changed to update `PI`'s precision.

One might argue: the same effect can be accomplished with an editor's search-and-replace feature. Just go ahead and use the literal value 3.14 throughout the code. When it must be changed, use the global search-and-replace to change every occurrence of 3.14 to 3.14159. This approach, however, is not a perfect solution. Some occurrences of this text string should *not* be replaced! Consider the statement:

```
gAlpha = 109.33381 * 93.14934 - entryAngle;
```

that could be buried somewhere in the thousands of lines of code. The editor's search-and-replace function would erroneously convert this statement to

```
gAlpha = 109.33381 * 93.14159934 - entryAngle;
```

This error is particularly insidious, since the intended literal value (the approximation for π) is not changed dramatically but only slightly. In addition, calculations based on `gAlpha`'s value, based in how the assignment statement was changed, may be off by only a small amount, so the bug may not be detected for some time. If the editor allowed the programmer to confirm before replacement, then the programmer must visually inspect each potential replacement, think about how the string is being used in context, and then decide whether to replace it or not. Obviously this process would be prone to human error. The proper use of symbolic constants eliminates this potential problem altogether.

8.7 Encapsulation

So far we have been using the `public` and `private` qualifiers for class members without much discussion about their usage. Recall `RYGTrafficLight` (Figure 7.4) from § 7.2. It nicely modeled traffic light objects and provided a methods so that clients can interact with the lights in well-defined but restricted ways. The state of each traffic light object is determined by exactly one integer: its `color` instance variable. This variable is well protected from direct access by clients:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> RYGTrafficLight t = new RYGTrafficLight();
> t.color = 182;
Error: This object cannot be modified
```

When developing complex systems, allowing indiscriminate access to an object's internals can be disastrous. It is all too easy for a careless, confused, or inept programmer to change an object's state in such a way as to corrupt the behavior of the entire system. A malicious programmer may intentionally tweak one or more objects to wreak havoc with the system. If the software system controls a medical device or military missile system, the results can be deadly.

Java provides several ways to protect the internals of an object from the outside world, but the simplest strategy is the one we have been using: fields and methods, generically referred to as class members, can be qualified as either `public` or `private`. These qualifiers mean the following:

- `public`—the class member is freely accessible by code in any class anywhere. `public` is the least restrictive qualifier.
- `private`—the class member is accessible only by code within the class itself. This is the most restrictive qualifier.

Said another way, `private` information is invisible to the outside world (clients of the class), but `public` information is freely accessible.² The compiler enforces the inaccessibility of `private` members. In traffic light objects, client code cannot modify `color` directly either accidentally or on purpose.

The following represent rules of thumb for using access specifiers:

- Data, that is, instance variables, generally should be `private`.
- Methods that constitute a service to be provided to clients should be `public`.
- Methods that serve other methods within the class but are not meant to be used outside the class should be `private`.

Why would a programmer intentionally choose to limit access to parts of an object? Restricting access obviously limits the client's control over the objects it creates. While this may appear to be a disadvantage at first glance, this access restriction actually provides a number of advantages:

- **Flexibility in implementation.** A class can be conceptually separated into two parts:

²It is not possible to protect an object from code within itself. If it appears that parts of a class should be protected from some of its methods, the class should be split up into multiple classes with suitable restrictions among the split up classes.

1. **Interface—the public part.** Clients see and can use the public part of an object. The public methods and public variables of a class constitute the *interface* of the class. A class's interface specifies *what* it does.
2. **Implementation—the hidden part.** Clients cannot see any `private` methods or `private` variables. Class developers are free to do whatever they want with the private parts of the class. A class's implementation specifies *how* it accomplishes what it needs to do.

We would like our objects to be black boxes: clients shouldn't need to know *how* they work, but clients rely on *what* they can do.

Many real-world objects follow this design philosophy. Consider a digital wristwatch. Its display gives its user the current time and, perhaps, date. It can produce different output in different modes; for example, elapsed time in stopwatch mode. It presents to its user only a few buttons for changing modes, starting and stopping stopwatches, and setting the time. *How* it does what it does is irrelevant to its user; its user is concerned with *what* it does. Its user risks great peril by opening the watch and looking at its intricate internal details. The user is meant to interact with the watch only through its interface—the display and buttons.

Similarly, an automobile presents an accelerator pedal to its user. The user knows that pushing the pedal makes the car go faster. That the pedal is connected to the fuel system (and possibly other systems, like cruise control) through a cable or other type of linkage is of concern only to the automotive designer or mechanic. Most drivers prefer to be oblivious to the under-the-hood details.

Changing the interface of a class disturbs client code that has been written to use objects of that class. For example, in the `RYGTrafficLight` class (Figure 7.4), the `draw()` method is `public` and is therefore part of the interface of the `RYGTrafficLight` class. If, after the `RYGTrafficLight` class has been made available to clients, the authors of the `RYGTrafficLight` class decide to eliminate the `draw()` method or rewrite it to accept parameters, then any existing client code that uses `draw()` according to its original definition will no longer be correct. We say the change in `RYGTrafficLight`'s interface *breaks* the client code. Class authors have no flexibility to alter the interface of a class once the class has been released for clients to use. On the other hand, altering the private information in a class will not break existing client code that uses that class, since private class information is invisible to clients. Therefore, a class becomes less resilient to change as more of its components become exposed to clients. To make classes as flexible as possible, which means maximizing the ability to make improvements to the class in the future, hide as much information as possible from clients.

- **Reducing programming errors.** Parts of a class that are `private` cannot be misused by client code since the client cannot see the `private` parts of a class. Properly restricting client access can make it impossible for client code to put objects into an undefined state. In fact, if a client can coax an object into an illegal state via the class interface, then the class has not been implemented properly. For example, recall the `color` instance variable of `RYGTrafficLight` (Figure 7.4). The decision to make `color` `private` means that the only way `color` can assume a value outside the range of 1...3 is if one of the `RYGTrafficLight`'s methods is faulty. Clients can never place a traffic light into an illegal state.
- **Hiding complexity.** Objects can provide a great deal of functionality. Even though a class may provide a fairly simple interface to clients, the services it provides may require a significant amount of complex code to accomplish their tasks. One of the challenges of software development is dealing with the often overwhelming complexity of the task. It is difficult, if not impossible, for one programmer to be able to comprehend at one time all the details of a large software system. Classes with well-designed interfaces and hidden implementations provide a means to reduce this complexity. Since `private` components of a class are hidden, their details cannot contribute to the complexity the client programmer must manage. The client programmer need not be concerned with exactly how an object works, but the details that make the object work are present nonetheless. The trick is exposing details only when necessary.

- **Class designer.** The class implementer must be concerned with the hidden implementation details of the class. The class implementer usually does not have to worry about the context in which the class will be used because the class may be used in many different contexts. From the perspective of the class designer, the complexity of the client code that may use the class is therefore eliminated.
- **Applications developer using a class.** The developer of the client code must be concerned with the details of the application code being developed. The application code will use objects. The details of the class these objects represent are of no concern to the client developers. From the perspective of the client code designer, the complexity of the code within classes used by the client code is therefore eliminated.

This concept of information hiding is called *encapsulation*. Details are exposed only to particular parties and only when appropriate. In sum, the proper use of encapsulation results in

- software that is more flexible and resilient to change,
- software that is more robust and reliable, and
- software development that is more easily managed.

8.8 Summary

- `System.out.println()` is used to display output to the console.
- `System.out.printf()` is used to display formatted output.
- `System.out.printf()` uses a format string and special control codes to justify and format numerical data.
- Coding conventions make it easier for humans to read source code, and ultimately humans must be able to easily read code in order to locate and fix bugs and extend programs.
- The three kinds of errors encountered by programmers are compile-time, runtime, and logic errors.
- The compiler notes compile-time errors; programs with compile-time errors cannot be compiled into bytecode.
- Runtime errors cannot be detected by the compiler but arise during the program's execution; runtime errors terminate the program's execution abnormally.
- Logic errors are usually the most difficult to fix. No error messages are given; the program simply does not run correctly.
- Symbolic constants should be used where possible instead of literal values. Symbolic constants make a program more readable, they shield the programmer from irrelevant information, and they lead to more maintainable code.
- Encapsulation hides code details in such a way so that details are only available to parties that need to know the details.
- The proper use of encapsulation results in code that is more flexible, robust, and easily managed.
- The public part of an object (or class) is called its interface.
- The private part of an object (or class) is called its implementation.

8.9 Exercises

1. What simple statement can be used to print the value of a variable named `x` on the console window?
2. The following control codes can be used in the format string of the `System.out.printf()` method call: `%d`, `%f`, `%s`, `%n`, `%%`. What do each mean?
3. Look at the documentation for `System.out.printf()` and find at least five control codes that are not mentioned in this chapter. What is the meaning of each of these codes?
4. Suppose `f` is a `double`. How can we print the value of `f` occupying 10 character positions, rounded to two decimal places, with leading zeroes? Provide the necessary statement and check with the following values: `-2.007`, `0.000003`, and `1234`.
5. Since the compiler is oblivious to it, why is consistent, standard source code formatting important?
6. List four generally accepted coding standards that are commonly used for formatting Java source.
7. Look at Sun's code convention website and list two suggested coding practices that are not mentioned in this chapter.
8. What are the three main kinds of errors that Java programs exhibit? Since programmers can more easily deal with some kinds of errors than others, rank the three kinds of errors in order of increasing difficulty to the programmer. Why did you rank them as you did?
9. What kind of error would result from the following programming mistakes, if any?
 - (a) Forgetting to end a statement in a method with a semicolon.
 - (b) Using a local variable in a method without declaring it.
 - (c) Printing a declared local variable in a method without initializing it.
 - (d) Using an instance variable in a method without declaring it.
 - (e) Printing a declared instance variable in a method without initializing it.
 - (f) Omitting a `return` statement in a non-void method.
 - (g) Integer division by zero.
 - (h) Floating-point division by zero.
 - (i) Calling an infinitely-recursive method, such as:

```
public int next(int n) {  
    return next(n + 1);  
}
```

First try to answer without using a computer and then check your answers by intentionally making the mistakes and seeing what errors, if any, actually occur.

10. List three advantages of using symbolic constants instead of literal values.
11. What is encapsulation, and how is it beneficial in programming?
12. Should instance variables generally be `public` or `private`?
13. How do you decide whether a method should be declared `public` or `private`?
14. How is a class's interface distinguished from its implementation?

Chapter 9

Class Members

An individual object has attributes that define the object's state. Some properties of objects are the same for all instances of its class. These common properties should not be stored in individual objects, but should be kept in a central location and shared by all instances of the class. This notion of class attributes generalizes to class methods, methods that need no instance-specific information to do their work.

9.1 Class Variables

`RYGTrafficLight` objects are automatically made red when created. It might be nice to have a constructor that accepts a color so the client can create a traffic light object of the desired color right away without having to call `change()`. Ideally

- The client should use symbolic constants like `RED` and `GREEN` instead of numbers like 1 and 2.
- The traffic light class author should control these constants so she has the freedom to change their values if necessary. For example, if the traffic light class author changes `RED` to be 2 instead of 1, clients should be oblivious, and the clients' code should continue to work without modification.
- The traffic light class author would make these constants available to clients.

If first glance, the solution seems simple enough—make the constants `RED`, `YELLOW`, and `GREEN`. in the `RYGTrafficLight` class `public` instead of `private`:

```
public final int RED    = 1;
public final int GREEN  = 2;
public final int YELLOW = 3;
```

Based on these definitions:

- The elements are constants (declared `final`), so clients cannot modify them.
- The elements are declared `public`, so clients can freely see their values.

Now we can overload the constructor providing a new one that accepts an integer argument:

```

public RYGTrafficLight(int initialColor) {
    if (initialColor >= RED && initialColor <= GREEN) {
        color = initialColor;
    } else {
        color = RED; // defaults to red
    }
}

```

and a client can use this constructor along with the color constants. Notice that the client can supply any integer value, but the constructor ensures that only valid color values are accepted. Arbitrary values outside the range of valid colors make a red traffic light.

So what is stopping us from using these constants this way? What happens when the client wants to create a traffic light that is initially green? If we write the code fragment that creates a green traffic light

```

RYGTrafficLight light = new RYGTrafficLight(???.GREEN);

```

what goes in the place of the ??? symbols? We need to replace the ??? with a RYGTrafficLight instance. As it now stands, we must have an object to access the RED, YELLOW, and GREEN constants. The problem is, the constants belong to RYGTrafficLight objects, and we cannot access them until we have at least one RYGTrafficLight object! This ultimately means we cannot create a traffic light object until we have created at least one traffic light object!

There is another problem with the color constants being members of each traffic light object. Every traffic light object needs to keep track of its own color, so color is an instance variable. There is no reason, however, why each traffic light needs to store its own copy of these three constant values. Suppose, for example, we were developing a program to model traffic flow through a section of a big city, and our system had to simultaneously manage 1,000 traffic light objects. The memory required to store these color constants in 1,000 objects would be

$$1,000 \text{ objects} \times 3 \frac{\text{ints}}{\text{object}} \times 4 \frac{\text{bytes}}{\text{int}} = 12,000 \text{ bytes}$$

This is unnecessary since these constants have the same value for every traffic light object. It would be convenient if these constants could be stored in one place and *shared* by all traffic light objects.

In Java, a class is more than a template or pattern for building objects. A class is itself an object-like entity. A class can hold data. This class data is shared among all objects of that class. The reserved word `static` is used to signify that a field is a *class variable* instead of an instance variable. Adding the `final` qualifier further makes it a *class constant*.

The option of class fields solves both of our problems with the color constants of the traffic light class. Declaring the constants this way:

```

public static final int RED    = 1;
public static final int GREEN  = 2;
public static final int YELLOW = 3;

```

(note the addition of the reserved word `static`) means that they can be accessed via the class itself, as in

```

RYGTrafficLight light = new RYGTrafficLight(RYGTrafficLight.GREEN);

```

The class name RYGTrafficLight is used to access the GREEN field instead of an instance name. If GREEN were not declared `static`, the expression RYGTrafficLight.GREEN would be illegal.

If we create 1,000 traffic light objects, they all share the three constants, so

$$3 \frac{\text{ints}}{\text{class}} \times 4 \frac{\text{bytes}}{\text{int}} = 12 \text{ bytes}$$

only 12 bytes of storage is used to store these constants no matter how many traffic light objects we create.

Given the declaration

```
public static final int RED    = 1;
```

we see, for example, RED is

- readable by any code anywhere (public),
- shared by all RYGTrafficLight objects (static),
- a constant (final),
- an integer (int), and
- has the value one (= 1).

We say that RED is a class constant. The following interactive session shows how the class fields can be used:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> RYGTrafficLight.RED
1
> RYGTrafficLight.YELLOW
2
> RYGTrafficLight.GREEN
3
```

Observe how we created no objects during the interactive session above (the new operator was not used). It is not necessary to create an object to access class fields.

Clients should be conscientious and use publicly defined class constants instead of literal values. Class designers provide public class constants as part of the contract they make with clients. Assuming our RYGTrafficLight class has been modified with the class constants described above, the following client code:

```
RYGTrafficLight light = new RYGTrafficLight(2);
```

makes a yellow traffic light object. The symbol YELLOW is guaranteed to stand for the yellow color, but YELLOW's actual value may change. If the maintainer of RYGTrafficLight updated the class by adding some colors (like black, for no lamps lit), YELLOW might be 3 instead of 2. The above code might then make a red light instead of the intended yellow light! The client's code is now broken, but it is the client's fault for using a literal constant (2) instead of the symbolic constant (RYGTrafficLight.YELLOW). If the statement above were part of a large complex program with many more similar statements throughout, the task to repair the code could be enormous. Just because you can see the value of a constant:

```
System.out.println(RYGTrafficLight.YELLOW);
```

does not mean it is wise to use its literal value.

9.2 Class Methods

Like fields, methods can be declared `static`. As with fields, this means the methods are *class methods*. All the methods we have seen so far have been *instance methods*. Instance methods must be called on behalf of an object, but a class method may be called on behalf of the class itself, and no instance is needed. In the `RYGTrafficLight` class, the `setColor()` method is an instance method, so

```
RYGTrafficLight t = new RYGTrafficLight(RYGTrafficLight.GREEN);
t.setColor(RYGTrafficLight.RED); // Legal
```

is valid Java, but

```
// Illegal! Compile-time error
RYGTrafficLight.setColor(RYGTrafficLight.RED);
```

is not. Since `setColor()` is an instance method, it cannot be called on behalf of the class; an object is required. Why is an object required? The code within `setColor()` modifies the `color` instance variable. Instance variables are not stored in the class; they are stored in objects (instances). You must call `setColor()` on behalf of an object, because the method needs to know which `color` variable (that is, the `color` variable in which object) to change.

While most of the methods we encounter are instance methods, class methods crop up from time to time. The `gcd()` method in `RationalNumber` (☞7.5) would be implemented better as a class method. `gcd()` gets all the information it needs from its parameters; it works the same regardless of the `RationalNumber` object with which it is called. No instance variables are used in `gcd()`. Any method that works independently of all instance information should be a class (static) method. The `gcd()` would better be written:

```
private static int gcd(int m, int n) {
    . . .
```

The JVM executes class methods more quickly than equivalent instance methods. Also, a class method cannot directly alter the state of the object upon which it is called since it has no access to the instance variables of that object. This restriction can actually simplify the development process since if an object's state becomes messed up (for example, a traffic light switching from red to yellow instead of red to green), a class method cannot be directly responsible for the ill-defined state. A developer would limit his search for the problem to instance methods.

A method that does not use any instance variables does not need to be an instance method. In fact, since a class method can be called on behalf of the class, it is illegal for a class method to attempt to access an instance variable or instance constant. It is also illegal for a class method to make an unqualified call to an instance method within the same class (because it could be accessing instance information indirectly). However, instance methods can freely access class variables and call class methods within the class.

9.3 Updated Rational Number Class

Armed with our knowledge of class fields and class methods, we can now enhance `RationalNumber` (☞7.5) to take advantage of these features. `Rational` (☞9.1) is our final version of a type representing mathematical rational numbers.

```
public class Rational {
```

```
private int numerator;
private int denominator;

// Some convenient constants
public static final Rational ZERO = new Rational(0, 1);
public static final Rational ONE = new Rational(1, 1);

// num is the numerator of the new fraction
// den is the denominator of the new fraction
public Rational(int num, int den) {
    if (den != 0) { // Legal fraction
        numerator = num;
        denominator = den;
    } else { // Undefined fraction changed to zero
        System.out.println("*** Notice: Attempt to create an "
            + "undefined fraction ***");
        numerator = 0;
        denominator = 1;
    }
}

// Returns the value of the numerator
public int getNumerator() {
    return numerator;
}

// Returns the value of the denominator
public int getDenominator() {
    return denominator;
}

// Provide a human-readable string for the fraction, like "1/2"
public String show() {
    return numerator + "/" + denominator;
}

// Compute the greatest common divisor for two integers m and n
// Uses Euclid's algorithm, circa 300 B.C.
private static int gcd(int m, int n) {
    if (n == 0) {
        return m;
    } else {
        return gcd(n, m % n);
    }
}

// Returns a new Rational consisting of the current
// fraction reduced to lowest terms
public Rational reduce() {
    int factor = gcd(numerator, denominator);
```

```

        return new Rational(numerator/factor,
                             denominator/factor);
    }

    // Returns a new Rational consisting of the sum of this
    // fraction and another fraction (other). The result is reduced
    // to lowest terms.
    public Rational add(Rational other) {
        int num1 = numerator * other.denominator,
            num2 = other.numerator * denominator;
        return new Rational(num1 + num2,
                             denominator * other.denominator).reduce();
    }

    // Returns a new Rational consisting of the product of this
    // fraction and another fraction (other). The result is reduced
    // to lowest terms.
    public Rational multiply(Rational other) {
        int num = numerator * other.numerator,
            den = other.denominator * denominator;
        return new Rational(num, den).reduce();
    }
}

```

Listing 9.1: Rational—updated version of the RationalNumber class

The list of enhancements consist of:

- The class

```
public class Rational {
```

is declared `public` which means it is meant for widespread use in many applications.

- The instance variables

```
private int numerator;
private int denominator;
```

are hidden from clients. Since none of the methods modify these instance variables, a `Rational` object is effectively *immutable*. (A Java `String` is another example of an immutable object.) The only way a client can influence the values of `numerator` and `denominator` in a particular fraction object is when it is created. After creation its state is fixed.

Note that even though the object may be immutable, an object reference is a variable and can be changed:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> Rational r = new Rational(1, 2);
> r.show()

```

```
"1/2"
> r = new Rational(2, 3);
> r.show()
"2/3"
```

Did the object `r` which was originally $\frac{1}{2}$ change to $\frac{2}{3}$? No, `Rational` objects are immutable and, therefore, cannot change. The object *reference* `r` was reassigned from pointing to object $\frac{1}{2}$ to the new object $\frac{2}{3}$. The original object $\frac{1}{2}$ is unchanged.

- The constructor now prints to the console a warning when a client attempts to create a `Rational` object with a denominator of zero.
- It is reasonable to allow clients to see the instance variables:

```
public int getNumerator() { . . .
public int getDenominator() { . . .
```

These methods return copies of the values of the instance variables; the instance variables themselves in a `Rational` object are safe from client tampering.

- Since zero (identity element for addition) and one (identity element for multiplication) are commonly used numbers, they have been provided as class constants:

```
public static final Rational ZERO = new Rational(0, 1);
public static final Rational ONE = new Rational(1, 1);
```

Based on their definition:

- `public`—they are freely available to clients
- `static`—they are class members and, therefore,
 - * they can be used even without creating a `Rational` object, and
 - * only one copy of each exists even if many `Rational` objects are created.
- `final`—they are constants and cannot be changed; for example, the `ONE` reference cannot be changed to refer to another `Rational` object (like $\frac{1}{2}$).
- `Rational`—they are references to `Rational` objects and so have all the functionality of, and can be treated just like, any other `Rational` object. For example,

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> Rational r = new Rational(1, 2);
> r = r.add(Rational.ZERO);
> System.out.println(r.show());
1/2
> r = r.add(Rational.ONE);
> System.out.println(r.show());
3/2
> r = r.add(Rational.ONE);
> System.out.println(r.show());
5/2
```

can be used without creating a `Rational` object

- The `gcd()` method

```
private static int gcd(int m, int n) {
```

is meant to be used internally by other methods within the class. The `reduce()` method uses `gcd()` to simplify a fraction. Our belief is that clients are interesting in reducing a fraction to lowest terms but are not concerned with the lower-level details of how this reduction is accomplished. We thus make `gcd()` private so clients cannot see or use it.

We further observe that `gcd()` does not access any instance variables. It performs its job strictly with information passed in via parameters. Thus we declare it `static`, so it is a class method. A class method is executed by the JVM more quickly than an equivalent instance method.

- The `show()`, `reduce()`, and `add()` methods are all made public since they provide a service to clients.
- This version provides a `multiply()` method.

9.4 An Example of Class Variables

We know that each object has its *own copy* of the instance variables defined in its class, but all objects of the same class *share* the class variables defined in that class. `Widget` (Figure 9.2) shows how class variables (not class constants) and instance variables might be used within the same class:

```
public class Widget {
    // The serial number currently available to be assigned to a new
    // widget object. This value is shared by all widget instances.
    private static int currentSerialNumber = 1;

    // The serial number of this particular widget.
    private int serialNumber;

    public Widget() {
        // Assign the currently available serial number to this
        // newly created widget, then update the currently available
        // serial number so it is ready for the next widget creation.
        serialNumber = currentSerialNumber++;
    }

    public void getSerialNumber() {
        System.out.println(serialNumber);
    }
}
```

Listing 9.2: `Widget`—Create widgets with unique, sequential serial numbers

The following interactive session makes some widgets and looks at their serial numbers:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> Widget w1 = new Widget(),
      w2 = new Widget(),
      w3 = new Widget(),
      w4 = new Widget(),
      w5 = new Widget();
> w1.getSerialNumber()
1
> w2.getSerialNumber()
2
> w3.getSerialNumber()
3
> w4.getSerialNumber()
4
> w5.getSerialNumber()
5
```

Exactly one copy of `currentSerialNumber` exists, but each `Widget` object has its own copy of `serialNumber`. The constructor uses this shared class variable to properly initialize the instance variable in each newly created object. Observe also that `serialNumber` is read-only; that is, the `getSerialNumber()` method can be used to read its value, but only the constructor can assign its value. Client code cannot change the `serialNumber` of an object to an arbitrary value. The `currentSerialNumber` class variable is totally inaccessible to the outside world, but it ensures that all widgets that a client creates will be numbered sequentially.

9.5 The `main()` Method

Any Java class may contain a method named `main()`, declared as

```
public static void main(String[] args) {
    /* Appropriate statements go here . . . */
}
```

We are familiar with all the parts of this method's declaration, except the square brackets (`[]`). The square bracket notation is used with arrays which we will not investigate until Chapter 20. Fortunately, since this method is special, we safely can ignore the `args` parameter for now. In fact, the overwhelming majority of Java programs containing such a `main()` method ignore the `args` parameter.

Classes with a `main()` method can be executed directly by the Java runtime environment (JRE), sometimes called the Java interpreter (not to be confused with the DrJava interpreter). We see that `main()` in a class method (`static`), so it can be executed without creating an object first. `VerySimpleJavaProgram` (Figure 9.3) shows how `main()` works:

```
public class VerySimpleJavaProgram {
    public static void main(String[] args) {
        System.out.println("This is very simple Java program!");
    }
}
```

Listing 9.3: VerySimpleJavaProgram—a very simple Java program

Once compiled, this code can be executed in the DrJava interpreter as:

Interactions

```
java VerySimpleJavaProgram
```

or similarly from a command shell in the operating system as

```
java VerySimpleJavaProgram
```

or by pressing the Run button in the DrJava button bar.

All of the parts of the above `main()` declaration must be present for things to work:

<code>public</code>	It must be accessible to the Java interpreter to be executed. Since the Java interpreter code is not part of our class, <code>main()</code> in our class must be <code>public</code> .
<code>static</code>	We as users must be able to run the program without having an object of that class available; hence, <code>main()</code> is a class (<code>static</code>) method.
<code>void</code>	Since the interpreter does not use any value that our <code>main()</code> might return, <code>main()</code> is required to return nothing (<code>void</code>).
<code>String[] args</code>	This allows extra information to be passed to the program when the user runs it. We have no need for this feature here for now, but we consider it later (§ 21.4).

9.6 Summary

- Class members (variables and methods) are specified with the `static` qualifier.
- A class variable is shared by all instances of that class.
- Class variables are stored in classes; instance variables are stored in instances (objects)
- Class methods may not access instance variables.
- Class methods may access class variables.
- Class methods may not make unqualified calls to instance methods.
- An instance method may be called only on behalf of an instance.
- A class method can be called on behalf of a class or on behalf of an instance.

9.7 Exercises

1. What reserved word denotes a class field or class method?

2. In the original version of `RYGTrafficLight` (Figure 7.4) (where the constants were instance fields, not class fields) if a client created 1,000 `RYGTrafficLight` objects, how much memory in bytes would the data (`color`, `RED`, `YELLOW`, `GREEN`) for all those objects require?
3. In the new version of `RYGTrafficLight` (Figure 7.4) (described in this chapter where the constants were class fields, not instance fields) if a client created 1,000 new style `RYGTrafficLight` objects, how much memory in bytes would the data (`color`, `RED`, `YELLOW`, `GREEN`) for all those objects require?
4. Given the class

```
public class Widget {
    public final int VALUE = 100;
    public Widget(int v) {
        System.out.println(v);
    }
}
```

does the following client code work?

```
Widget w = new Widget(Widget.VALUE);
```

If it works, what does it do? If it does not work, why does it not work?

5. Why is it not possible to make `color` a class variable in `RYGTrafficLight` (Figure 7.4)?
6. What determines whether a method should be a class method or an instance method?
7. If clients can see the values of public class constants, why should clients avoid using the constants' literal values?
8. Enhance `Rational` (Figure 9.1) by adding operations for:
 - subtraction
 - division
 - computing the reciprocal
9. Enhance `Rational` (Figure 9.1) show that the `show()` method properly displays mixed numbers as illustrated by the following interactive sequence:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> Rational r = new Rational(5, 2);
> System.out.println(r.show());
2 1/2
> r = new Rational(4, 1);
> System.out.println(r.show());
4
> r = new Rational(11, 4);
> System.out.println(r.show());
2 3/4
> r = new Rational(1, 4);
> System.out.println(r.show());
1/4
```

10. Next item . . .

11. Consider the following class definition:

```
public class ClassDef {  
    static public double value;  
    static public int number;  
    public double quantity;  
    public int amount;  
}
```

Suppose 10,000 instances of `ClassDef` were created. How much memory would be consumed by the variables held by all the objects?

12. What is the exact structure of the `main()` method that allows a class to be executed as a program.

13. Why must `main()` be declared `static`?

14. Create a class named `Point` which can be used to create mathematical point objects. Such objects:

- (a) have (x,y) coordinates which are doubles
- (b) have a `distance()` method to compute the distance to another point.
- (c)

15. Augment the `RationalNumber` class so that `RationalNumber` objects can:

- (a) multiply
- (b) subtract
- (c) etc.

16. We have seen how `reduce()` (an instance method) in `Rational` calls `gcd()` (a class method) to assist its work. What happens if `gcd()` calls an instance method of `Rational`? Explain what happens.

Chapter 10

Composing Objects

The design of `RYGTrafficLight` (Figure 7.4) can be improved. The core functionality of traffic light objects is illuminating the correct lamp at the correct time, and `RYGTrafficLight` does this well. Some clients, however, may desire a different display. In most modern traffic lights the lamps are arranged vertically instead of horizontally. Perhaps instead of colored lights, printed or spoken commands are to be issued, such as “Stop,” “Prepare to stop,” and “Proceed.” The sequence of commands would correspond to lamp colors, so the core logic of `RYGTrafficLight` would still be appropriate, but the output to the user would be much different.

Traffic light objects have two distinct responsibilities:

- The **core logic** manages the state of the traffic light, properly sequencing the signals.
- The **presentation** displays the state of the light in a manner appropriate to the client.

We will devise separate classes to address these responsibilities. In software design, the part of the system responsible for the core logic is called the *model*. The presentation part is commonly referred to as the *view*.

10.1 Modeling Traffic Light Logic

Since the core logic can be presented in a variety of ways, the names we use for variables and methods will be more generic than those used in `RYGTrafficLight` (Figure 7.4). `TrafficLightModel` (Figure 10.1) is a new version of our traffic light class that incorporates class constants:

```
public class TrafficLightModel {
    // Constants used to describe the states of the traffic light
    public static final int OFF      = 0;
    public static final int STOP    = 1;
    public static final int CAUTION = 2;
    public static final int GO      = 3;

    // The light's current state, one of the constants listed above
    private int state;
}
```

```

// Creates a light with a given initial color
public TrafficLightModel(int initialState) {
    setState(initialState); // Ensures a legal state
}

// Creates a light that is initially in the "stop" state
public TrafficLightModel() {
    this(STOP); // Default light is STOP for safety's sake
}

// Returns the current color of the light
public int getState() {
    return state;
}

// Determines if a given integer maps to a legal state
public boolean isLegalState(int potentialState) {
    return potentialState == OFF || potentialState == STOP
        || potentialState == CAUTION || potentialState == GO;
}

// Sets the light's state. The state is unchanged if
// the integer value passed does not map to a legal state.
public void setState(int newState) {
    if (isLegalState(newState)) {
        state = newState;
    } else { // For safety's sake any other int value makes STOP
        state = STOP;
    }
}

// Changes the light's state to the next legal state in its normal cycle.
public void change() {
    if (getState() == STOP) {
        setState(GO);
    } else if (getState() == CAUTION) {
        setState(STOP);
    } else if (getState() == GO) {
        setState(CAUTION);
    } // else the signal remains the same; i.e., OFF lights stay OFF
}
}

```

Listing 10.1: TrafficLightModel—final version of the traffic light class

The differences from RYGTrafficLight (Figure 7.4) include:

- The class's name is TrafficLightModel. The name implies that a TrafficLightModel object will model the behavior of a traffic light, but it will not contribute to the view.

- Colors have been replaced with descriptive state names: `STOP`, `CAUTION`, `GO`. These states can be interpreted by the view component as colored lights, voice commands, or whatever else as needed.
- A new state has been added: `OFF`. If added to `RYGTrafficLight` (Figure 7.4), `OFF` would correspond to the color black, meaning no lamps lit. This state is useful to make flashing traffic lights, like `STOP`, `OFF`, `STOP`, `OFF`, ..., which could be visualized as a flashing red light.
- The drawing functionality has been removed since `TrafficLightModel` objects provide only the model and not the view.
- Clients can see that state of the object via the `getState()` method. This is essential since the view needs to be able to see the state of the model in order to render the traffic light correctly.
- In a limited way clients can modify the model's state without using the `change()` method. The `setState()` method allows a client to set the state instance variable to any valid state value, but it does not allow the client to set the state to an arbitrary value, like 82.
- The Boolean method `isLegalState()` determines which integers map to legal states and which do not.
- The `change()` method makes use of two other methods—`getState()` and `setState()`—to examine and update the state instance variable. We could have written `change()` similar to our earlier traffic light code:

```
public void change() {
    if (state == STOP) {
        state = GO;
    } else if (state == CAUTION) {
        state = STOP;
    } else if (state == GO) {
        state = CAUTION;
    } // else the signal remains the same; i.e., OFF lights stay OFF
}
```

This code is simpler to understand (which is good), but the new version that uses `getState()` and `setState()` offers an important advantage. Any access to the state instance variable must go through `getState()` or `setState()`. For `setState()` the advantage is clear since `setState()` uses `isLegalState()` to ensure an invalid integer is not assigned to the state variable. Consider for a moment a more sophisticated scenario. Suppose we wish to analyze how our traffic light models are used. We will keep track of an object's state by logging each time the state variable is examined or modified. We can use an integer instance variable named `readCount` that is incremented each time state is examined (read). Similarly, we can use an instance variable named `writeCount` that is incremented each time state is modified (written). If we limit access to the state variable to the `getState()` and `setState()` methods, the code to keep track of those statistics can be isolated to those two methods. If we instead allow methods such as `change()` to directly examine or modify the state variable, we would have to duplicate our logging code everywhere we allow such direct access. These presents several disadvantages:

- We have to type in more code. When directly assigning state we must remember to also increment `writeCount`:

```
state = GO;
writeCount++;
```

It is simpler and cleaner instead to use

```
setState(GO);
```

and let `setState()` take care of the accounting. The shortest, simplest code that accomplishes the task is usually desirable. More complex code is harder to understand and write correctly.

- We must ensure that the logging code is being used consistently in all places that it should appear. Is it really the same code in all the places that it must appear, or are there errors in one or more of the places it appears? It is easy to mistakenly write within a method

```
state = GO;
readCount++;
```

Here, `writeCount` should have been incremented instead of `readCount`. Errors like this one can be difficult sometimes to track down.

- We must ensure that we do not forget to write the logging code in all the places it should appear. One occurrence in a method of a statement like

```
state = GO;
```

without the associated logging code will lead to errors in our analysis. If we instead use `setState()`, the accounting is automatically handled for us.

While the code presented here is not as sophisticated as this logging scenario, we could mistakenly write code in one of our methods like

```
state = newValue;
```

thinking that `newValue` is in the range 0–3. If it is not, the traffic light model would enter an undefined state. If we use `setState()` instead, it is impossible for the traffic light model to be in an undefined state.

The `TrafficLightModel` class represents objects that are more complex than any objects we have devised up to this point. Neither constructor assigns the `state` instance variable directly. One constructor defers to the other one via the `this` call (see § 7.1); the other constructor uses the `setState()` method to assign `state`. Since `setState()` contains the logic to make sure the integer passed in is legitimate, it makes sense not to duplicate the effort of checking the legitimacy in two different places within the class. Within the `setState()` method, another method, `isLegalState()`, is used to actually check to see if the proposed integer is valid.

The various methods within `TrafficLightModel` interact with the class constants and instance variable to perform their tasks. Figure 10.1 illustrates the relationships of the pieces of a `TrafficLightModel` object.

Biological cells are units that have complex interworkings. Cells have parts such as nuclei, ribosomes, and mitochondria that much all work together using complex processes to allow the cell to live and perform its function within the larger framework of a tissue. The tissue is a building block for an organ, and a number of organs are assembled into organisms.

In like manner, a software object such a `TrafficLightModel` instance is itself reasonably complex (although not nearly as complex as a living cell!). The component relationship diagram in Figure 10.1 almost gives the appearance of the biochemical pathways within a living cell. Like cells, programming objects themselves can be quite complex and can be used to build up more complex software structures called *components* which are used to build software systems.

By itself a `TrafficLightModel` object is not too useful. A `TrafficLightModel` object must be paired with a view object to be a fully functional traffic light.

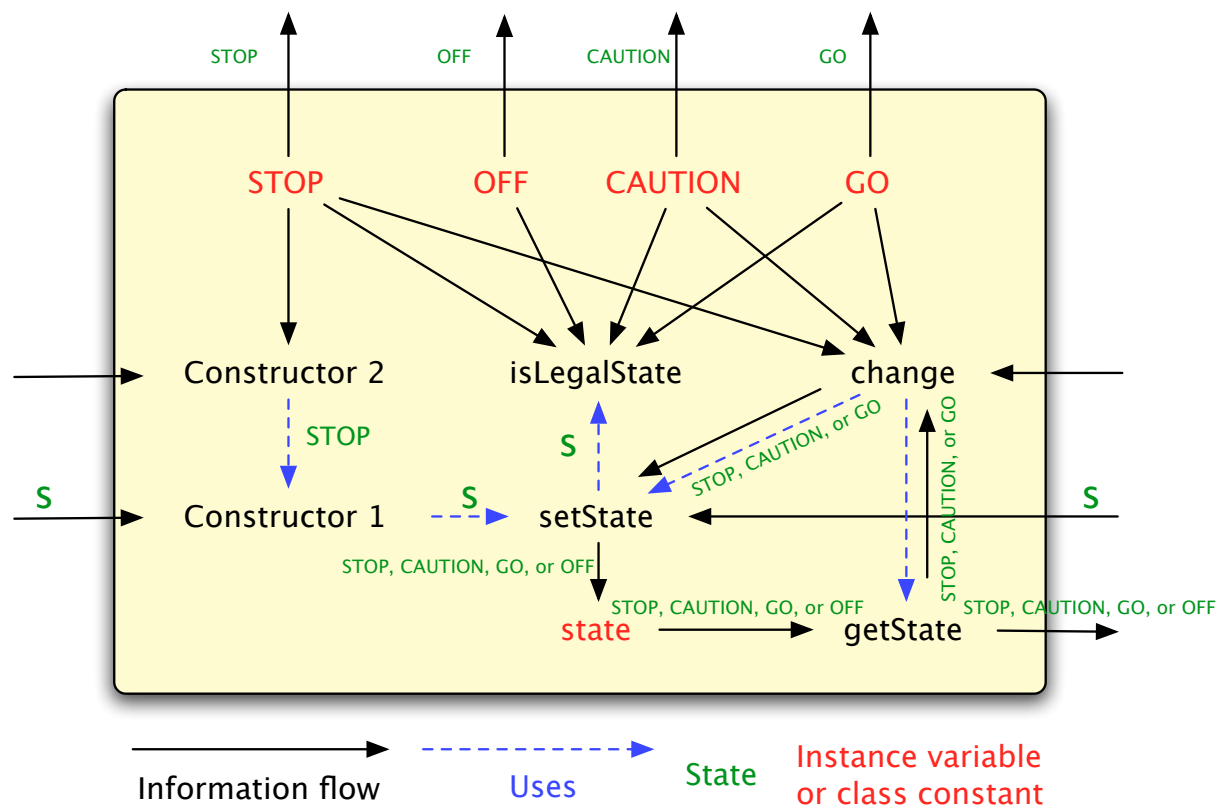


Figure 10.1: Component relationships within traffic light model objects. The solid arrows indicate information flow. Class constants can be read (all arrows flow out) but cannot be modified (no arrows flow in). The state variable can be read only by `getState()` (only arrow out) and modified only by the `setState()` method (only arrow in). Arrows that enter or leave the traffic light model object indicate information provided by or available to client code. The dashed arrows show when a method employs the services of another method; for example, `setState()` uses `isLegalState` in order to complete its task. Similarly, `change()` calls `getState()` and can call `setState()`.

10.2 Textual Visualization of a Traffic Light

`TextTrafficLight` (Figure 10.2) uses `TrafficLightModel` (Figure 10.1) to display a text-based visualization of the state of a traffic light:

```
public class TextTrafficLight {
    // Provides core functionality
    private TrafficLightModel light;

    // Create a text traffic light from a traffic light model
    public TextTrafficLight(TrafficLightModel lt) {
        light = lt;
    }

    // Returns the current state
    public int getState() {
        return light.getState();
    }
}
```

```

// Sets the state
public void setState(int newState) {
    light.setState(newState);
}

// Change can change the model's state, if the model allows it
public void change() {
    light.change();
}

// Renders each lamp
public String drawLamps() {
    if (light.getState() == TrafficLightModel.STOP) {
        return "(R) ( ) ( )"; // Red stop light
    } else if (light.getState() == TrafficLightModel.CAUTION) {
        return "( ) (Y) ( )"; // Yellow caution light
    } else if (light.getState() == TrafficLightModel.GO) {
        return "( ) ( ) (G)"; // Green go light
    } else {
        return "( ) ( ) ( )"; // Otherwise off
    }
}

// Renders the lamps within a frame
public String show() {
    return "[" + drawLamps() + "]";
}
}

```

Listing 10.2: TextTrafficLight—a text-based traffic light visualization

The following interactive session demonstrates how a client can make and use a TextTrafficLight object:

Interactions
<pre> Welcome to DrJava. Working directory is /Users/rick/java > t = new TextTrafficLight(new TrafficLightModel(TrafficLightModel.STOP)); > System.out.println(t.show()); [(R) () ()] > t.change(); System.out.println(t.show()); [() () (G)] > t.change(); System.out.println(t.show()); [() (Y) ()] > t.change(); System.out.println(t.show()); [(R) () ()] > t.change(); System.out.println(t.show()); [() () (G)] > t.change(); System.out.println(t.show()); [() (Y) ()] > t.change(); System.out.println(t.show()); </pre>

```
[ (R) ( ) ( ) ]
```

In `TextTrafficLight` (Figure 10.2):

- The lone instance variable, `light`, is an object reference to a `TrafficLightModel` object. We say a `TextTrafficLight` object is *composed of* a `TrafficLightModel` object. A `TextTrafficLight` object uses a `TrafficLightModel` and expects it to manage the traffic light's state.
- The `getState()`, `setState()`, and `change()` methods do nothing interesting themselves except call the equivalent methods of the model. This technique is called *delegation*. We say a `TextTrafficLight` object delegates the work of its method to its contained `TrafficLightModel` object. This concept of composition with delegation is common in OO programming.

What advantage does this composition provide? We have separated the implementation of a traffic light's behavior (what colors it can assume and the sequence of colors in a cycle) from its appearance. This decoupling is good, since given a `TrafficLightModel` we can create a new visualization without worrying about the details of how the light operates. The visualization works with the model through the model's public interface and does not know anything about the model's implementation.

To see the payoff of this decoupling, we will create a new visualization of our traffic light. In our `TextTrafficLight` objects the lamps are arranged horizontally, left to right, red to green. Most traffic lights used today in the US arrange their lamps vertically, top to bottom, red to green. In `VerticalTextLight` (Figure 10.3) we model such a vertical traffic light:

```
public class VerticalTextLight {
    // Provides core functionality
    private TrafficLightModel light;

    // Create a text traffic light from a traffic light model
    public VerticalTextLight(TrafficLightModel lt) {
        light = lt;
    }

    // Returns the current state
    public int getState() {
        return light.getState();
    }

    // Sets the state
    public void setState(int newState) {
        light.setState(newState);
    }

    // Change changes the model's state
    public void change() {
        light.change();
    }

    // Renders each lamp
    public String drawLamps() {
```

```

    if (light.getState() == TrafficLightModel.STOP) {
        return "|(R)|\n|(|)\n|(|)\n"; // Red stop light
    } else if (light.getState() == TrafficLightModel.CAUTION) {
        return "|(|)\n|(Y)|\n|(|)\n"; // Yellow caution light
    } else if (light.getState() == TrafficLightModel.GO) {
        return "|(|)\n|(|)\n|(G)|\n"; // Green go light
    } else {
        return "|(|)\n|(|)\n|(|)\n"; // Off otherwise
    }
}

// Renders the lamps within a frame
public String show() {
    return "+---+\n" + drawLamps() + "+---+";
}
}

```

Listing 10.3: VerticalTextLight—an alternate text-based traffic light visualization

The "\n" sequence is an *escape sequence* that represents the newline character. This forces the output to appear on the next line. The following interactive session illustrates:

Interactions
<pre> Welcome to DrJava. Working directory is /Users/rick/java > t = new VerticalTextLight(new TrafficLightModel(TrafficLightModel.STOP)); > System.out.println(t.show()); +---+ (R) () () +---+ > t.change(); System.out.println(t.show()); +---+ () () (G) +---+ > t.change(); System.out.println(t.show()); +---+ () (Y) () +---+ > t.change(); System.out.println(t.show()); +---+ (R) () () </pre>

```
+---+
```

An even more impressive visualization can be achieved using *real* graphics. We will defer the better example until § 12.3, after the concept of inheritance is introduced.

VerticalTextLight (§ 10.3) and TextTrafficLight (§ 10.2) both leverage the TrafficLightModel (§ 10.1) code, the only difference is the presentation. The design is more *modular* than, say, RYGTrafficLight (§ 7.4), that combines the logic and the presentation into one class. By comparison, the RYGTrafficLight class is *non-modular*. TextTrafficLight and VerticalTextLight objects are visual shells into which we “plug in” our TrafficLightModel object to handle the logic.

Modular design, as we have just demonstrated, can lead to *code reuse*. Code reuse is the ability to use existing code in a variety of applications. A *component* is a reusable software unit—a class—that can be used as is in many different software contexts. In our case, TrafficLightModel is a component that can be plugged into different traffic light visualization objects. Component-based software development seeks to streamline programming by making it easier to build complex systems by assembling components and writing as little original code as possible. Component-based software development thus consists largely of component construction and combining components to build the overall application.

10.3 Intersection Example

Traffic lights are used to control traffic flow through intersections. We can model an intersection as an object. Our intersection object is composed of traffic light objects, and these lights must be coordinated to ensure the safe flow of traffic through the intersection.

An intersection object is provided (via a constructor) four traffic light objects in the order north, south, east, and west. The intersection object’s `change()` method first does a simple consistency check and then, if the check was successful, changes the color of the lights.

Intersection (§ 10.4) is the blueprint for such an intersection object.

```
public class Intersection {
    // Four traffic lights for each of the four directions
    private TextTrafficLight northLight;
    private TextTrafficLight southLight;
    private TextTrafficLight eastLight;
    private TextTrafficLight westLight;

    // The constructor requires the client to provide the traffic
    // light objects
    public Intersection(TextTrafficLight north, TextTrafficLight south,
                       TextTrafficLight east, TextTrafficLight west) {
        northLight = north;
        southLight = south;
        eastLight = east;
        westLight = west;
        // Check to see if the configuration of lights is valid
        if (!isValidConfiguration()) {
            System.out.println("Illegal configuration");
        }
    }
}
```

```

}

// Checks to see that there are no conflicts with the current
// light colors. The first two parameters and the last two
// parameters are opposing lights. Callers must ensure the
// proper ordering of the parameters.
public boolean isValidConfiguration() {
    // Sanity check:
    // Opposite lights should be the same color for this
    // controller to work properly; adjacent lights should not be
    // the same color. (Other controllers might specify
    // different rules.)
    if (northLight.getState() != southLight.getState()
        || eastLight.getState() != westLight.getState()
        || northLight.getState() == eastLight.getState()) {
        // Some conflict exists; for safety's sake, make them all
        // red
        northLight.setState(TrafficLightModel.STOP);
        southLight.setState(TrafficLightModel.STOP);
        eastLight.setState(TrafficLightModel.STOP);
        westLight.setState(TrafficLightModel.STOP);
        return false; // Error condition, return false for failure
    } else {
        return true;
    }
}

// Properly synchronizes the changing of all the
// lights in the intersection, then it (re)displays
// the intersection.
public void change() {
    if (northLight.getState() == TrafficLightModel.CAUTION
        || eastLight.getState() == TrafficLightModel.CAUTION) {
        // If any light is yellow, all need to change
        northLight.change();
        southLight.change();
        eastLight.change();
        westLight.change();
    } else if (northLight.getState() != TrafficLightModel.STOP) {
        // If north/south lights are NOT red, change them
        northLight.change();
        southLight.change();
    } else if (eastLight.getState() != TrafficLightModel.STOP) {
        // If east/west lights are NOT red, change them
        eastLight.change();
        westLight.change();
    }
}

// Draw the contents of the intersection in two dimensions; e.g.,

```

```

//              (North)
//              (West)      (East)
//              (South)
public void show() {
    System.out.println("              " + northLight.show());
    System.out.println();
    System.out.println(westLight.show() + "              " +
                       eastLight.show());
    System.out.println();
    System.out.println("              " + southLight.show());
}
}

```

Listing 10.4: Intersection—simulates a four-way intersection of simple traffic lights

The following interactive session exercises an Intersection object:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> i = new Intersection(new TextTrafficLight
    (new TrafficLightModel(TrafficLightModel.STOP)),
    new TextTrafficLight
    (new TrafficLightModel(TrafficLightModel.STOP)),
    new TextTrafficLight
    (new TrafficLightModel(TrafficLightModel.GO)),
    new TextTrafficLight
    (new TrafficLightModel(TrafficLightModel.GO)));
> i.show();
              [(R) ( ) ( )]

[( ) ( ) (G)]              [( ) ( ) (G)]
              [(R) ( ) ( )]

> i.change(); i.show();
              [(R) ( ) ( )]

[( ) (Y) ( )]              [( ) (Y) ( )]
              [(R) ( ) ( )]

> i.change(); i.show();
              [( ) ( ) (G)]

[(R) ( ) ( )]              [(R) ( ) ( )]
              [( ) ( ) (G)]

> i.change(); i.show();
              [( ) (Y) ( )]

```

```

[(R) ( ) ( )]          [(R) ( ) ( )]

          [( ) (Y) ( )]
> i.change(); i.show();
          [(R) ( ) ( )]

[( ) ( ) (G)]          [( ) ( ) (G)]

          [(R) ( ) ( )]
> i.change(); i.show();
          [( ) ( ) (R)]

[( ) (Y) ( )]          [( ) (Y) ( )]

          [( ) ( ) (R)]

```

The sequence of signals allows for safe and efficient traffic flow through the modeled intersection.

In Intersection (Figure 10.4) we see:

- Each Intersection object contains four TextTrafficLight objects, one for each direction:

```

private TextTrafficLight northLight;
private TextTrafficLight southLight;
private TextTrafficLight eastLight;
private TextTrafficLight westLight;

```

These statements declare the instance variable names, but do not create the objects; recall that objects are created with the `new` operator.

These TextTrafficLight objects are components being reused by the Intersection class.

- The constructor is responsible for initializing each traffic light object.

```

public Intersection(TextTrafficLight north, TextTrafficLight south,
                   TextTrafficLight east, TextTrafficLight west) {
    northLight = north;
    . . .
}

```

It is the client's responsibility to create each light in the proper state. We did so in the rather lengthy initialization statement at the beginning on the interactive session above. The constructor does check to see if the initial state of the system of lights is acceptable by calling `isValidConfiguration()`. In the case of an illegal initial configuration of lights (for example, all the lights are green, or north and west both green), all the lights are made red. This leads to a defunct but safe intersection.

- The `change()` method coordinates the changing of each individual light.
- The `show()` method draws the lights (actually asks the individual lights to draw themselves) in locations corresponding to their location.

IntersectionMain (Figure 10.5) builds a program around Intersection (Figure 10.4):

```
import java.util.Scanner;

public class IntersectionMain {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        Intersection intersection
            = new Intersection(new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.STOP)),
                new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.STOP)),
                new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.GO)),
                new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.GO)));

        while (true) {    // infinite loop
            scan.nextLine();    // Wait for return key press
            intersection.change();
            intersection.show();
        }
    }
}
```

Listing 10.5: IntersectionMain—simulates a four-way intersection of simple traffic lights

The `scan.nextLine()` waits for the user to enter a string and press **Return**. If the user simply presses **Return**, an empty string (no characters) is returned by `nextLine()`, and the program's execution continues. The statement beginning

```
while (true) {
```

forms an infinite loop; code within its body executes over and over again continuously. Loops are covered in more detail in Chapter 17. This infinite loop ensures that the cycle continues until the user terminates the program.

10.4 Summary

- Variables within objects can be references to other objects.
- Object composition allows complex objects to be built up from simpler objects.
- Delegation is when a complex object implements an operation by calling the corresponding operation of its contained object. The corresponding operation in the contained object has some conceptual similarity to the container's operation.
- Modular design separates functionality into focused, individual classes that can be used in a variety of software contexts.

- Code reuse is the ability to use existing code (classes) in a variety of applications.
-

10.5 Exercises

1. What is meant by *object composition*? How is it accomplished?
2. What is meant by *composition with delegation*? How is it accomplished?
3. What is meant by *modular design*? How is it accomplished?
4. What advantages are provided by modular design?
5. What is *code reuse*, and why is it important?
6. How does `Intersection` (Figure 10.4) behave when the initial configuration is north = green, east = green, south = red, and west = red?
7. What issues are there with `VerticalTextLight` (Figure 10.3) objects work with the existing `Intersection` (Figure 10.4) code?
8. Not only can a model be plugged into multiple views. A new model can be made and plugged into an existing view. To verify this, develop a new model that makes a flashing red light. That is, calling the `change()` method will cycle the colors red, black, red, black, ... Test your new class with the existing `TextTrafficLight` (Figure 10.2) and `VerticalTextLight` (Figure 10.3).

Chapter 11

Inheritance and Polymorphism

Composition is one example of code reuse. We have seen how classes can be composed with other classes to make more sophisticated classes. In this chapter we will see how classes can be reused in a different way. Using inheritance, a programmer can make a new class out of an existing class, thereby providing a way to create objects with enhanced behavior.

11.1 Inheritance

Recall `TextTrafficLight` (▮10.2). `TextTrafficLight` objects simulate the cycling behavior of simple, real-world traffic lights. In `Intersection` (▮10.4) we modeled a four-way intersection synchronizing the operation of four `TextTrafficLight` objects. Suppose the traffic at our intersection has grown, and it has become difficult for vehicles to make left-hand turns because the volume of oncoming traffic is too great. The common solution is to add a left turn arrow that allows opposing traffic to turn left without interference of oncoming traffic. Clearly we need an enhanced traffic light that provides such a left turn arrow. We could copy the existing `TrafficLightModel` source code, rename it to `TurnLightModel`, and then commence modifying it so that it provides the enhance functionality. We then could copy the `TextTrafficLight` code and modify it so that it can display a left turn arrow. This is not unreasonable given the size of our classes, but there are some drawbacks to this approach:

- Whenever code is copied and modified there is the possibility of introducing an error. It is always best, as far as possible, to leave working code untouched.
- If the code is copied and a latent error is discovered and fixed in the original code, the copied code should be repaired as well. The maintainers of the original code may not know who is using copies of their code and, therefore, cannot notify all concerned parties of the change.

Object-oriented languages provide a mechanism that addresses both of these issues. New classes can be built from existing classes using a technique known as *inheritance*, or *subclassing*. This technique is illustrated in `TurnLightModel` (▮11.1):

```
public class TurnLightModel extends TrafficLightModel {
    // Add a new state indicating left turn
    public static int LEFT_TURN = 1000;
```

```

// Creates a light in the given initial state
public TurnLightModel(int initialState) {
    super(initialState);
}

// Add LEFT_TURN to the set of valid states
public boolean isLegalState(int potentialState) {
    return potentialState == LEFT_TURN
        || super.isLegalState(potentialState);
}

// Changes the state of the light to its next state in its normal cycle.
// Properly accounts for the turning state.
public void change() {
    int currentState = getState();
    if (currentState == LEFT_TURN) {
        setState(GO);
    } else if (currentState == STOP) {
        setState(LEFT_TURN);
    } else {
        super.change();
    }
}
}

```

Listing 11.1: TurnLightModel—extends the TrafficLightModel class to make a traffic light with a left turn arrow

In TurnLightModel (Figure 11.1):

- The reserved word `extends` indicates that TurnLightModel is being derived from an existing class—TrafficLightModel. We can say this in English various ways:
 - TurnLightModel is a *subclass* of TrafficLightModel, and TrafficLightModel is the *superclass* of TurnLightModel.
 - TurnLightModel is a *derived class* of TrafficLightModel, and TrafficLightModel is the *base class* of TurnLightModel.
 - TurnLightModel is a *child class* of TrafficLightModel, and TrafficLightModel is the *parent class* of TurnLightModel.
- By virtue of being a subclass, TurnLightModel inherits all the characteristics of the TrafficLightModel class. This has several key consequences:
 - While you do not see a state instance variable defined within the TurnLightModel class, all TurnLightModel objects have such an instance variable. The state variable is inherited from the superclass. Just because all TurnLightModel objects have a state variable does not mean the code within their class can access it directly—state is still private to the superclass TrafficLightModel. Fortunately, code within TurnLightModel can see state via the inherited `getState()` method and change state via `setState()`.

- While you see neither `getState()` nor `setState()` methods defined in the `TurnLightModel` class, all `TurnLightModel` objects have these methods at their disposal since they are inherited from `TrafficLightModel`.
- `TurnLightModel` inherits the state constants `OFF`, `STOP`, `CAUTION`, and `GO` and adds a new one, `LEFT_TURN`. `LEFT_TURN`'s value is defined so as to not coincide with the previously defined state constants. We can see the values of `OFF`, `STOP`, `CAUTION`, and `GO` because they are publicly visible, so here we chose 1,000 because it is different from all of the inherited constants.
- The constructor appears to be calling an instance method named `super`:

```
super(initialState);
```

In fact, `super` is a reserved word, and when it is used in this context it means call the superclass's constructor. Here, `TurnLightModel`'s constructor ensures the same initialization activity performed by `TrafficLightModel`'s constructor will take place for `TurnLightModel` objects. In general, a subclass constructor can include additional statements following the call to `super()`. If a subclass constructor provides any statements of its own besides the call to `super()`, they must follow the call of `super()`.

- `TurnLightModel` provides a revised `isLegalState()` method definition. When a subclass redefines a superclass method, we say it *overrides* the method. This version of `isLegalState()` expands the set of integer values that map to a valid state. `isLegalState()` returns true if the supplied integer is equal to `LEFT_TURN` or is approved by the superclass version of `isLegalState()`. The expression:

```
super.isLegalState(potentialState)
```

looks like we are calling `isLegalState()` with an object reference named `super`. The reserved word `super` in this context means execute the superclass version of `isLegalState()` on behalf of the current object. Thus, `TurnLightModel`'s `isLegalState()` adds some original code (checking for `LEFT_TURN`) and reuses the functionality of the superclass. It in essence does what its superclass does plus a little extra.

Recall that `setState()` calls `isLegalState()` to ensure that the client does not place a traffic light object into an illegal state. `TurnLightModel` does not override `setState()`—it is inherited as is from the superclass. When `setState()` is called on a pure `TrafficLightModel` object, it calls the `TrafficLightModel` class's version of `isLegalState()`. By contrast, when `setState()` is called on behalf of a `TurnLightModel` object, it calls the `TurnLightModel` class's `isLegalState()` method. This ability to “do the right thing” with an object of a given type is called *polymorphism* and will be addressed in § 11.4.

- The `change()` method inserts the turn arrow state into its proper place in the sequence of signals:
 - red becomes left turn arrow,
 - left turn arrow becomes green, and
 - all other transitions remain the same (the superclass version works fine)

Like `isLegalState()`, it also reuses the functionality of the superclass via the `super` reference.

Another interesting result of inheritance is that a `TurnLightModel` object will work fine in any context that expects a `TrafficLightModel` object. For example, a method defined as

```
public static void doTheChange(TrafficLightModel tlm) {
    System.out.println("The light changes!");
    tlm.change();
}
```

obviously accepts a `TrafficLightModel` reference as an actual parameter, because its formal parameter is declared to be of type `TrafficLightModel`. What may not be so obvious is that the method will also accept a `TurnLightModel` reference. Why is this possible? A subclass inherits all the capabilities of its superclass and usually adds some more. This means anything that can be done with a superclass object can be done with a subclass object (and the subclass object can probably do more). Since any `TurnLightModel` object can do at least as much as a `TrafficLightModel`, the `tlm` parameter can be assigned to `TurnLightModel` just as easily as to a `TrafficLightModel`. The `doTheChange()` method calls `change()` on the parameter `tlm`. `tlm` can be an instance of `TrafficLightModel` or *any subclass* of `TrafficLightModel`. We say an *is a* relationship exists from the `TurnLightModel` class to the `TrafficLightModel` class. This is because any `TurnLightModel` object *is a* `TrafficLightModel` object.

In order to see how our new turn light model works, we need to visualize it. Again, we will use inheritance and derive a new class from an existing class, `TextTrafficLight`. `TextTurnLight` (▮ 11.2) provides a text visualization of our new turn light model:

```
public class TextTurnLight extends TextTrafficLight {
    // Note: constructor requires a turn light model
    public TextTurnLight(TurnLightModel lt) {
        super(lt); // Calls the superclass constructor
    }

    // Renders each lamp
    public String drawLamps() {
        // Draw non-turn lamps
        String result = super.drawLamps();
        // Draw the turn lamp properly
        if (getState() == TurnLightModel.LEFT_TURN) {
            result += " (<) ";
        } else {
            result += " ( ) ";
        }
        return result;
    }
}
```

Listing 11.2: `TextTurnLight`—extends the `TextTrafficLight` class to make a traffic light with a left turn arrow

`TextTurnLight` (▮ 11.2) is a fairly simple class. It is derived from `TextTrafficLight` (▮ 10.2), so it inherits all of `TextTrafficLight`'s functionality, but the differences are minimal:

- The constructor expects a `TurnLightModel` object and passes it to the constructor of its superclass. The superclass constructor expects a `TrafficLightModel` reference as an actual parameter. A `TurnLightModel` reference is acceptable, though, because a `TurnLightModel` *is a* `TrafficLightModel`.

- In `drawLamps()`, a `TextTurnLight` object must display four lamps instead of only three. This method renders all four lamps. The method calls the superclass version of `drawLamps()` to render the first three lamps:

```
super.drawLamps();
```

and so the method needs only draw the last (turn) lamp.

Notice that the `draw()` method, which calls `drawLamps()`, is not overridden. The subclass inherits and uses `draw()` as is, because it does not need to change how the “frame” is drawn.

The constructor requires clients to create `TextTurnLight` objects with only `TurnLightModel` objects. A client may not create a `TextTurnLight` with a simple `TrafficLightModel`:

```
// This is illegal
TextTurnLight lt
    = new TextTurnLight(new TrafficLightModel
                        (TrafficLightModel.RED));
```

The following interaction sequences demonstrates some of the above concepts. First, we will test the light’s cycle:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> TextTurnLight lt = new TextTurnLight
    (new TurnLightModel
     (TrafficLightModel.STOP));

> System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]
```

All seems to work fine here. Next, let us experiment with this *is a* concept. Reset the Interactions pane and enter:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> TextTrafficLight lt = new TextTurnLight
```

```

                (new TurnLightModel
                  (TrafficLightModel.STOP));

> System.out.println(lt.show());
[(R) ( ) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[( ) ( ) ( ) (<)]
> lt.change(); System.out.println(lt.show());
[( ) ( ) (G) ( )]
> lt.change(); System.out.println(lt.show());
[( ) (Y) ( ) ( )]
> lt.change(); System.out.println(lt.show());
[(R) ( ) ( ) ( )]

```

Notice that here the variable `lt`'s declared type is `TextTrafficLight`, not `TextTurnLight` as in the earlier interactive session. No error is given because a `TextTurnLight` object (created by the new expression) *is a* `TextTrafficLight`, and so it can be assigned legally to `light`. Perhaps Java is less picky about assigning objects? Try:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> Intersection light = new TextTurnLight
                        (new TurnLightModel
                          (TrafficLightModel.STOP));

Error: Bad types in assignment

```

Since no superclass/subclass relationship exists between `Intersection` and `TextTurnLight`, there is no *is a* relationship either, and the types are not assignment compatible. Furthermore, the *is a* relationship works only one direction. Consider:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> TextTurnLight lt2 = new TextTrafficLight
                        (new TrafficLightModel
                          (TrafficLightModel.STOP));

ClassCastException: lt2

```

All `TurnLightModels` are `TrafficLightModels`, but the converse is not true. As an illustration, all apples are fruit, but it is not true that all fruit are apples.

While inheritance may appear to be only a clever programming trick to save a little code, it is actually quite useful and is used extensively for building complex systems. To see how useful it is, we will put our new kind of traffic lights into one of our existing intersection objects and see what happens. First, to simplify the interactive experience, we will define `TestIntersection` (Figure 11.3), a convenience class for making either of the two kinds of intersections:

```

public class TestIntersection {
    public static Intersection makeSimple() {

```



```

        return new Intersection(
            new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.STOP)),
            new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.STOP)),
            new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.GO)),
            new TextTrafficLight
                (new TrafficLightModel(TrafficLightModel.GO)));
    }
    public static Intersection makeTurn() {
        return new Intersection(
            new TextTurnLight
                (new TurnLightModel(TrafficLightModel.STOP)),
            new TextTurnLight
                (new TurnLightModel(TrafficLightModel.STOP)),
            new TextTurnLight
                (new TurnLightModel(TrafficLightModel.GO)),
            new TextTurnLight
                (new TurnLightModel(TrafficLightModel.GO)));
    }
}

```

Listing 11.3: TestIntersection—provides some convenience methods for creating two kinds of intersections

Both methods are class (static) methods, so we need not explicitly create a TestIntersection object to use the methods. The following interactive session creates two different kinds of intersections:

Interactions	
Welcome to DrJava. Working directory is /Users/rick/java	
> simple = TestIntersection.makeSimple();	
> simple.show();	
[(R) () ()]	
[() () (G)]	[() () (G)]
[(R) () ()]	
> simple.change(); simple.show();	
[(R) () ()]	
[() (Y) ()]	[() (Y) ()]
[(R) () ()]	
> simple.change(); simple.show();	
[() () (G)]	
[(R) () ()]	[(R) () ()]
[() () (G)]	

```

> simple.change(); simple.show();
      [ ( ) (Y) ( ) ]

[ (R) ( ) ( ) ]           [ (R) ( ) ( ) ]

      [ ( ) (Y) ( ) ]
> simple.change(); simple.show();
      [ (R) ( ) ( ) ]

[ ( ) ( ) (G) ]           [ ( ) ( ) (G) ]

      [ (R) ( ) ( ) ]
> simple.change(); simple.show();
      [ (R) ( ) ( ) ]

[ ( ) (Y) ( ) ]           [ ( ) (Y) ( ) ]

      [ (R) ( ) ( ) ]
> turn = TestIntersection.makeTurn();
> turn.show();
      [ (R) ( ) ( ) ( ) ]

[ ( ) ( ) (G) ( ) ]           [ ( ) ( ) (G) ( ) ]

      [ (R) ( ) ( ) ( ) ]
> turn.change(); turn.show();
      [ (R) ( ) ( ) ( ) ]

[ ( ) (Y) ( ) ( ) ]           [ ( ) (Y) ( ) ( ) ]

      [ (R) ( ) ( ) ( ) ]
> turn.change(); turn.show();
      [ ( ) ( ) ( ) (<) ]

[ (R) ( ) ( ) ( ) ]           [ (R) ( ) ( ) ( ) ]

      [ ( ) ( ) ( ) (<) ]
> turn.change(); turn.show();
      [ ( ) ( ) (G) ( ) ]

[ (R) ( ) ( ) ( ) ]           [ (R) ( ) ( ) ( ) ]

      [ ( ) ( ) (G) ( ) ]
> turn.change(); turn.show();
      [ ( ) (Y) ( ) ( ) ]

[ (R) ( ) ( ) ( ) ]           [ (R) ( ) ( ) ( ) ]

      [ ( ) (Y) ( ) ( ) ]
> turn.change(); turn.show();

```

```

                [(R) ( ) ( ) ( )]

[( ) ( ) ( ) (<)]                [( ) ( ) ( ) (<)]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [(R) ( ) ( ) ( )]

[( ) ( ) (G) ( )]                [( ) ( ) (G) ( )]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [(R) ( ) ( ) ( )]

[( ) (Y) ( ) ( )]                [( ) (Y) ( ) ( )]

                [(R) ( ) ( ) ( )]
> turn.change(); turn.show();
                [( ) ( ) ( ) (<)]

[(R) ( ) ( ) ( )]                [(R) ( ) ( ) ( )]

                [( ) ( ) ( ) (<)]

```

Notice that our original `Intersection` class was not modified at all, yet it works equally as well with `TextTurnLight` objects! This is another example of the “magic” of inheritance. A `TextTurnLight` object can be treated exactly like a `TextTrafficLight` object, yet it behaves in a way that is appropriate for a `TextTurnLight`, not a `TextTrafficLight`. A `TextTrafficLight` object draws three lamps when asked to `show()` itself, while a `TextTurnLight` draws four lamps. This is another example of polymorphism (see § 11.4).

11.2 Protected Access

We have seen how client access to the instance and class members of a class are affected by the `public` and `private` specifiers:

- Elements declared `public` within a class are freely available to code in any class to examine and modify.
- Elements declared `private` are inaccessible to code in other classes. Such private elements can only be accessed and/or influenced by `public` methods provided by the class containing the private elements.

Sometimes it is desirable to allow special privileges to methods within subclasses. Java provides a third access specifier—`protected`. A `protected` element cannot be accessed by other classes in general, but it can be accessed by code within a subclass. Said another way, `protected` is like `private` to non-subclasses and like `public` to subclasses.

Class designers should be aware of the consequences of using `protected` members. The `protected` specifier weakens encapsulation (see Section 8.7). Encapsulation ensures that the internal details of a class cannot be disturbed by client code. Clients should be able to change the state of an object only through the public methods provided. If these public methods are correctly written, it will be impossible for client code to put an object into an undefined

or illegal state. When fields are made `protected`, careless subclassers may write methods that misuse instance variables and place an object into an illegal state. Some purists suggest that `protected` access never be used because the potential for misuse is too great.

Another issue with `protected` is that it limits how superclasses can be changed. Anything `public` becomes part of the class's interface to all classes. Changing public members can break client code. Similarly, anything `protected` becomes part of the class's interface to its subclasses, so changing protected members can break code within subclasses.

Despite the potential problems with the `protected` specifier, it has its place in class design. It is often convenient to have some information or functionality shared only within a family (inheritance hierarchy) of classes. For example, in the traffic light code, the `setState()` method in `TrafficLightModel` (Figure 10.1) might better be made `protected`. This would allow subclasses like `TurnLightModel` (Figure 11.1) to change a light's state, but other code would be limited to making a traffic light with a specific initial color and then alter its color only through the `change()` method. This would prevent a client from changing a green light immediately to red without the caution state in between. The turn light code, however, needs to alter the basic sequence of signals, and so it needs special privileges that should not be available in general.

Since encapsulation is beneficial, a good rule of thumb is to reveal as little as possible to clients and subclasses. Make elements `protected` and/or `public` only when it would be awkward or unworkable to do otherwise.

11.3 Visualizing Inheritance

The Unified Modeling Language is a graphical, programming language-independent way of representing classes and their associated relationships. The UML can quickly communicate the salient aspects of the class relationships in a software system without requiring the reader to wade through the language-specific implementation details (that is, the source code). The UML is a complex modeling language that covers many aspects of system development. We will limit our attention to a very small subset of the UML used to represent class relationships.

In the UML, classes are represented by rectangles. Various kinds of lines can connect one class to another, and these lines represent relationships among the classes. Three relationships that we have seen are composition, inheritance, and dependence.

- **Composition.** An `Intersection` object is composed of four `TextTrafficLight` objects. Each `TextTrafficLight` object manages its own `TrafficLightModel` object, and a `TextTurnLight` contains a `TurnLightModel`. The UML diagram shown in Figure 11.1 visualizes these relationships. A solid line from one class to another with a diamond at one end indicates composition. The end with the diamond is connected to the container class, and the end without the diamond is connected to the contained class. In this case we see that an `Intersection` object contains `TextTrafficLight` objects. A number at the end of the line indicates how many objects are contained. (If no number is provided, the number 1 is implied.) We see that each intersection is composed of four traffic lights, while each light has an associated model.

The composition relationship is sometimes referred to as the *has a* relationship; for example, a text traffic light *has a* traffic light model.

- **Inheritance.** `TurnLightModel` is a subclass of `TrafficLightModel`, and `TextTrafficLight` is a subclass of `TextTurnLight`. The inheritance relationship is represented in the UML by a solid line with a triangular arrowhead as shown in Figure 11.2. The arrow points from the subclass to the superclass.

We have already mentioned that the inheritance relationship represents the *is a* relationship. The arrow points in the direction of the *is a* relationship.

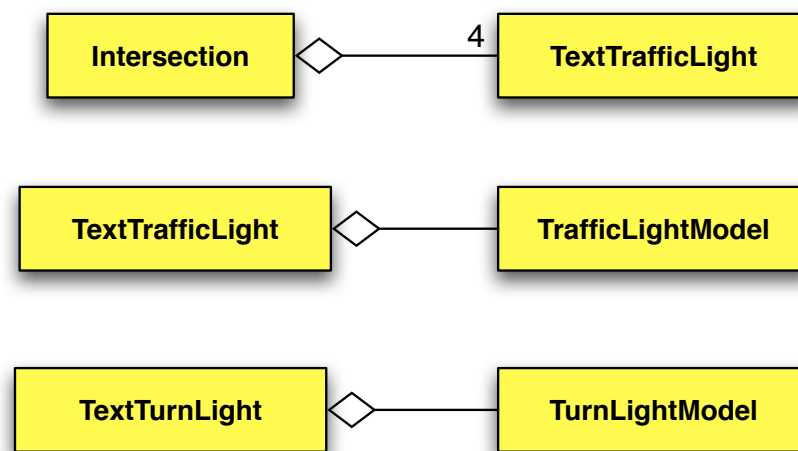
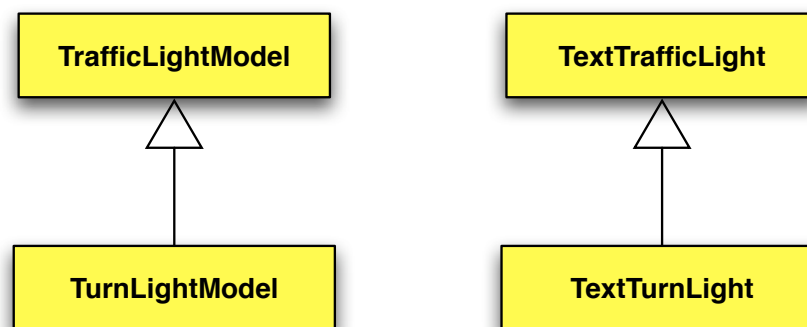
Figure 11.1: UML diagram for the composition relationships involved in the `Intersection` class

Figure 11.2: UML diagram for the traffic light inheritance relationship

- **Dependence.** We have used dependence without mentioning it explicitly. Objects of one class may use objects of another class without extending them (inheritance) or declaring them as fields (composition). Local variables and parameters represent temporary dependencies. For example, the `TestIntersection` class uses the `Intersection`, `TrafficLightModel`, `TextTrafficLight`, `TurnLightModel`, and `TextTurnLight` classes within its methods (local objects), but neither inheritance nor composition are involved. We say that `TestIntersection` depends on these classes, because if their interfaces change, those changes may affect `TestIntersection`. For example, if the maintainers of `TextTurnLight` decide that its constructor should accept an integer state instead of a `TurnLightModel`, the change would break `TestIntersection`. Currently `TestIntersection` creates a `TextTurnLight` with a `TurnLightModel`, not an integer state.

A dashed arrow in a UML diagram illustrates dependency. The label `<<uses>>` indicates that a `TestIntersection` object uses the other object in a transient way. Other kinds of dependencies are possible. Figure 11.3 shows the dependency of `TestIntersection` upon `Intersection`.

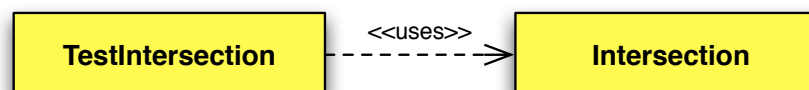


Figure 11.3: UML diagram for the test intersection dependencies

Figure 11.4 shows the complete diagram of participating classes.

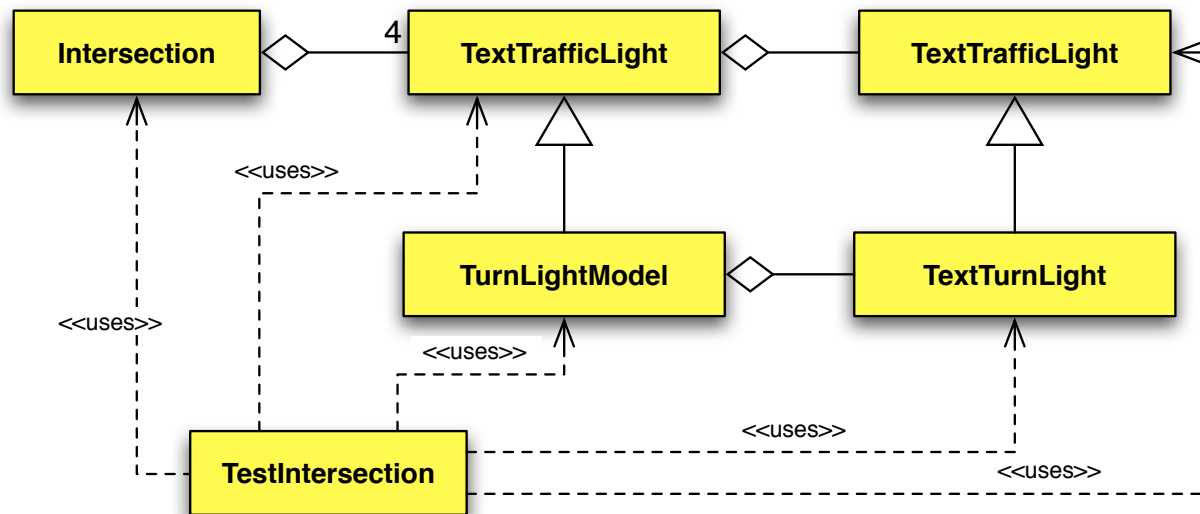


Figure 11.4: the complete UML diagram for the classes used in `TestIntersection`

11.4 Polymorphism

How does the code within `Intersection`'s `show()` method decide which of the following ways to draw a red light?

```
[ (R) ( ) ( ) ]
```

or

```
[ (R) ( ) ( ) ( ) ]
```

Nothing within `Intersection`'s `show` method reveals any distinction. It does not use any `if/else` statements to select between one form or another. The `show()` method does the right thing based on the exact kind of traffic light that it is asked to render. Can the compiler determine what to do when it compiles the source code for `Intersection`? The compiler is powerless to do so, since the `Intersection` class was developed and tested *before* the `TurnLightModel` class was ever conceived!

The compiler generates code that at runtime decides which `show()` method to call based on the actual type of the light. The burden is, in fact, on the object itself. The expression

```
northLight.show()
```

is a request to the `northLight` object to draw itself. The `northLight` object draws itself based on the `show()` method in its class. If it is really a `TextTrafficLight` object, it executes the `show()` method of the `TextTrafficLight` class; if it is really a `TextTurnLight` object, it executes the `show()` method of the `TextTurnLight` class.

This process of executing the proper code based on the exact type of the object when the *is a* relationship is involved is called *polymorphism*. Polymorphism is what makes the `setState()` methods work as well. Try to set a plain traffic light to the `LEFT_TURN` state, and then try to set a turn traffic light to `LEFT_TURN`:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> plain = new TextTrafficLight(new TrafficLightModel(TrafficLightModel.STOP));
> turn = new TextTurnLight(new TurnLightModel(TrafficLightModel.STOP));
> plain.setState(TurnLightModel.LEFT_TURN);
> System.out.println(plain.show());
[(R) ( ) ( )]
> turn.setState(TurnLightModel.LEFT_TURN);
> System.out.println(turn.show());
[( ) ( ) ( ) (<)]
```

Remember, `setState()` was not overridden. The same `setState()` code is executed for `TurnLightModel` objects as for `TrafficLightModel` objects. The difference is what happens when `setState()` calls the `isLegalState()` method. For `TrafficLightModel` objects, `TrafficLightModel`'s `isLegalState()` method is called; however, for `TurnLightModel` objects, `TurnLightModel`'s `isLegalState()` method is invoked. We say that `setState()` calls `isLegalState()` polymorphically. `isLegalState()` polymorphically “decides” whether `LEFT_TURN` is a valid state. The “decision” is easy though; call it on behalf of a pure `TrafficLightModel` object, and says “no,” but call it on behalf of a `TurnLightModel` object, and it says “yes.”

Polymorphism means that given the following code:

```
TextTrafficLight light;

// Initialize the light somehow . . .

light.show();
```

we cannot predict whether three lamps or four lamps will be displayed. The code between the two statements may assign `light` to a `TextTrafficLight` object or a `TextTurnLight` depending on user input (§ 8.3), a random number (§ 13.6, time (§ 13.2), or any of hundreds of other criteria.

11.5 Extended Rational Number Class

`Rational` (Figure 9.1) is a solid, but simple, class for rational number objects. Addition and multiplication is provided, but what if we wish to subtract or divide fractions? Without subclassing clients could do all the dirty work themselves:

- Subtracting is just adding the opposite:

$$\frac{a}{b} - \frac{c}{d} = \frac{a}{b} + \left(-1 \times \frac{c}{d}\right)$$

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> Rational f1 = new Rational(1, 2),
    f2 = new Rational(1, 4);
> // f3 = f1 - f2 = f1 + (-1) * f2
> Rational f3 = f1.add(new Rational(-1, 1).multiply(f2));
> f3.show()
"1/4"
> f2 = new Rational(1, 3);
```

```
> f3 = f1.add(new Rational(-1, 1).multiply(f2));
> f3.show()
"1/6"
```

Since $f1 = \frac{1}{2}$ and $f2 = \frac{1}{4}$, the statement

```
Rational f3 = f1.add(new Rational(-1, 1).multiply(f2));
```

results in

$$f3 = \frac{1}{2} + \left(\frac{-1}{1} \times \frac{1}{4} \right) = \frac{2}{4} - \frac{1}{4} = \frac{1}{4}$$

and for $f2 = \frac{1}{3}$, the statement

```
f3 = f1.add(new Rational(-1, 1).multiply(f2));
```

results in

$$f3 = \frac{1}{2} + \left(\frac{-1}{1} \times \frac{1}{3} \right) = \frac{3}{6} - \frac{2}{6} = \frac{1}{6}$$

- Dividing is multiplying by the inverse:

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c}$$

Interactions

```
> // 1/2 divided by 2/3 = 3/4
> f1.show()
"1/2"
> f2 = new Rational(2, 3);
> f2.show()
"2/3"
> f3 = f1.multiply(new Rational(f2.getDenominator(),
                                f2.getNumerator()));
> f3.show()
"3/4"
```

The problem with this approach is that it is messy and prone to error. It would be much nicer to simply say:

```
f3 = f1.subtract(f2);
```

and

```
f3 = f1.divide(f2);
```

but these statements are not valid if `f1` is a `Rational` object. What we need is an extension of `Rational` that supports the desired functionality. `EnhancedRational` (▮ 11.4) is such a class.


```

public class EnhancedRational extends Rational {
    // num is the numerator of the new fraction
    // den is the denominator of the new fraction
    // Work deferred to the superclass constructor
    public EnhancedRational(int num, int den) {
        super(num, den);
    }
    // Returns this - other reduced to lowest terms
    public Rational subtract(Rational other) {
        return add(new Rational(-other.getNumerator(),
                                other.getDenominator()));
    }
    // Returns this / other reduced to lowest terms
    // (a/b) / (c/d) = (a/b) * (d/c)
    public Rational divide(Rational other) {
        return multiply(new Rational(other.getDenominator(),
                                    other.getNumerator()));
    }
}

```

Listing 11.4: EnhancedRational—extended version of the Rational class

With EnhancedRational (Listing 11.4) subtraction and division are now more convenient:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> EnhancedRational f1 = new EnhancedRational(1, 2),
    f2 = new EnhancedRational(1, 4);
> Rational f3 = f1.subtract(f2);
> f3.show()
"1/4"
> f3 = f1.divide(f2);
> f3.show()
"2/1"

```

11.6 Multiple Superclasses

In Java it is not possible for a class to have more than one superclass. Some languages like C++ and Smalltalk do support multiple superclasses, a concept called *multiple inheritance*.

Even though a class may not have more than one superclass, it may have any number of subclasses, including none.

11.7 Summary

- Inheritance allows us to derive a new class from an existing class.

- The original class is called the superclass, and the newly derived class is called the subclass.
- Other terminology often used instead of superclass/subclass base class/derived class, and parent class/child class.
- The subclass inherits everything from its superclass and usually adds more capabilities.
- The reserved word `extends` is used to subclass an existing class.
- Subclasses can redefine inherited methods; the process is called overriding the inherited method.
- Subclass methods can call superclass methods directly via the `super` reference.
- Constructors can invoke the superclass constructor using `super` in its method call form.
- A subclass inherits everything from its superclass, including private members (variables and methods), but it has no extra privileges accessing those members than any other classes.
- Subclass objects have an *is a* relationship with their superclass; that is, if `Y` is a subclass of `X`, an instance of `Y` *is a* `Y`, and an instance of `Y` *is a* `X` also.
- A subclass object may be assigned to a superclass reference; for example, if `Y` is a subclass of `X`, an instance of `Y` may be assigned to a variable of type `X`.
- A superclass object may **not** be assigned to a subclass reference; for example, if `Y` is a subclass of `X`, an instance whose exact type is `X` may not be assigned to a variable of type `Y`.
- The Unified Modeling Language (UML) uses special graphical notation to represent classes, composition, inheritance, and dependence.
- Polymorphism executes a method based on an object's exact type, not simply its declared type.
- A class may not have more than one superclass, but it may have zero or more subclasses.

11.8 Exercises

1. Suppose the variable `lt` is of type `TextTrafficLight` and that it has been assigned properly to an object. What will be printed by the statement:

```
System.out.println(lt.show());
```

2. What does it mean to override a method?
3. What is polymorphism? How is it useful?
4. If `TurnLightModel` (¶ 11.1) inherits the `setState()` method of `TrafficLightModel` (¶ 10.1) and uses it as is, why and how does the method behave differently for the two types of objects?
5. May a class have multiple superclasses?
6. May a class have multiple subclasses?

7. Devise a new type of traffic light that has `TrafficLightModel` (§10.1) as its superclass but does something different from `TurnLightModel` (§11.1). One possibility is a flashing red light. Test your new class in isolation, and then create a view for your new model. Your view should be oriented horizontally so you can test your view with the existing `Intersection` (§10.4) code.
8. Derive `VerticalTurnLight` from `VerticalTextLight` (§10.3) that works like `TextTurnLight` (§11.2) but that displays its lamps vertically instead of horizontally. Will you also have to derive a new model from `TurnLightModel` (§11.1) to make your `VerticalTurnLight` work?

Chapter 12

Simple Graphics Programming

Graphical applications are more appealing to users than text-based programs. When designed properly, graphical applications are more intuitive to use. From the developers perspective, graphical programs provide an ideal opportunity to apply object-oriented programming concepts. Java provides a wealth of classes for building graphical user interfaces (GUIs). Because of the power and flexibility provided by these classes experienced developers can use them to create sophisticated graphical applications. On the other hand, also because of the power and flexibility of these graphical classes, writing even a simple graphics program can be daunting for those new to programming—especially those new to graphics programming. For those experienced in building GUIs, Java simplifies the process of developing good GUIs. Good GUIs are necessarily complex. Consider just two issues:

- Graphical objects must be positioned in an attractive manner even when windows are resized or the application must run on both a PC (larger screen) and a PDA or cell phone (much smaller screen). Java graphical classes simplify the programmer's job of laying out of screen elements within an application.
- The user typically can provide input in several different ways: via mouse clicks, menu, dialog boxes, and keystrokes. The program often must be able to respond to input at any time, not at some predetermined place in the program's execution. For example, when a statement calling `nextInt()` on a `Scanner` object (§ 8.3) is executed, the program halts and waits for the user's input; however, in a graphical program the user may select any menu item at any time. We say that graphics programs are *event driven* instead of program driven. Java's event classes readily support event-driven program development.

While Java's infrastructure is a boon to seasoned GUI programmers, at this time Java has no standard graphics classes that make graphics programming more accessible to beginners. The unfortunate result is that beginners must learn a number of different principles (including Java's layout management and event model), before writing even the simplest interactive graphical programs.

In this chapter we introduce some simplified graphics classes that beginners can use to write interactive graphical programs. These classes are built from standard Java classes, but they insulate the programmer from much of the complexity of Java GUI development.

12.1 2D Graphics Concepts

A *graphic* is an image drawn on a display device, usually the screen. The smallest piece of a graphic is called a *pixel*, short for *picture element*. The size of a pixel is fixed by the hardware resolution of the display device. A graphic is

rendered on a rectangular drawing surface. Modern operating systems use a window-based interface, so the drawing surface for a graphical application is usually the area inside the frame of a window assigned to that application. Each rectangular drawing surface has a coordinate system. The coordinate system is slightly different from the Cartesian coordinate system used in mathematics:

- the origin, $(0,0)$, is located at the left-top corner of the rectangular region, and
- the y axis points down, meaning that the y values increase as you go down.

Figure 12.1 illustrates:

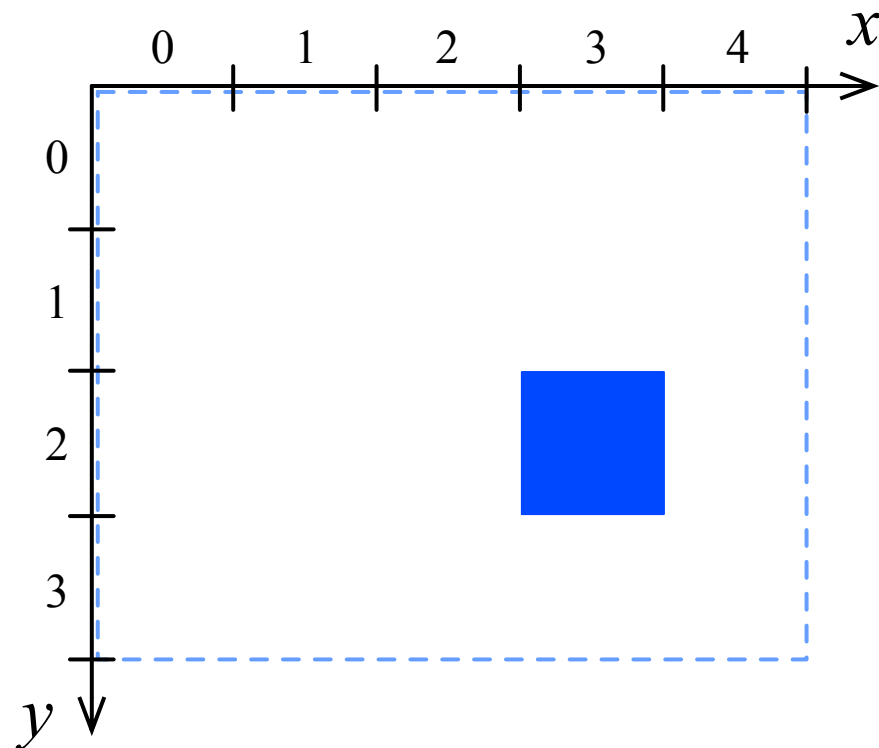


Figure 12.1: The graphics coordinate system. The small square is a highly magnified view of a pixel at coordinate $(3,2)$. The dotted frame represents the bounds of the rectangular drawing surface which is five pixels wide and four pixels tall.

12.2 The Viewport Class

In our simplified graphics classes a graphical window is represented by an object of type `Viewport`.¹ Graphics can be displayed within viewports, and viewports can receive input events from the user, such as mouse movement and clicking. The following interactive sequence creates and manipulates a simple viewport:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> import edu.southern.computing.oopj.Viewport;
```

¹The name `Viewport` was chosen to be especially distinct from the standard Java window types: `Frame` and `JFrame`. `Viewport` is implemented as a subclass of `javax.swing.JFrame`.

```
> w = new Viewport("Very Simple Window", 100, 100, 300, 200);  
> w.setSize(400, 550);  
> w.setLocation(50, 100);
```

In this brief interactive sequence:

- The first statement imports the necessary class definition. Since the package name does not begin with `java...`, we know immediately that this is not a standard class.
- Variable `w` is assigned to a new `Viewport` object. The parameters to the constructor are, in order:
 - the string title to appear in the window's (viewport's) title bar,
 - the x coordinate of the left-top corner of the viewport,
 - the y coordinate of the left-top corner of the viewport,
 - the width of the viewport in pixels, and
 - the height of the viewport in pixels.
- The `setSize()` method resizes the window to the specified width and height. The window's left top corner does not move.
- The `setLocation()` method repositions the window so that its left top corner is moved to the specified (x,y) location. The window's size is unchanged.

Observe that in addition to calling the `setSize()` and `setLocation()` methods, the user can reposition and resize the window using the mouse or other pointing device.

The `Viewport` class has a `draw()` method that determines the visual contents of a viewport. The default `draw()` method does nothing, hence our empty viewport in the example above. In order to draw within a viewport, we must create a subclass of `Viewport` and override the `draw()` method.

A viewport provides methods that can be used within the `draw()` method to render primitive shapes. The shapes include rectangles, ellipses, polygons, lines, points, and strings. Table 12.1 lists some of the more frequently used methods:

Some Methods of the <code>drawingprimitives</code> Class	
<code>void setColor(Color c)</code>	Sets the current drawing color to <code>c</code> . <code>c</code> can be one of <code>BLACK</code> , <code>WHITE</code> , <code>RED</code> , <code>BLUE</code> , color constants defined for rectangular drawing surfaces.
<code>void drawLine(int x1, int y1, int x2, int y2, Color c)</code>	Draws a one-pixel wide line from <code>(x1,y1)</code> to <code>(x2,y2)</code> . If the last parameter is omitted, the current drawing color is used.
<code>void drawPoint(int x, int y, Color c)</code>	Plots a point (a single pixel) at a given <code>(x,y)</code> position. If the last parameter is omitted, the current drawing color is used.
<code>void drawRectangle(int left, int top, int width, int height)</code>	Draws the outline of a rectangle given the position of its left-top corner and size.
<code>void fillRectangle(int left, int top, int width, int height, Color c)</code>	Renders a solid of a rectangle given the position of its left-top corner and size. If the last parameter is omitted, the current drawing color is used.
<code>void drawOval(int left, int top, int width, int height)</code>	Draws the outline of an ellipse given the position of the left-top corner and size of its bounding rectangle.
<code>void fillOval(int left, int top, int width, int height, Color c)</code>	Renders a solid of a ellipse given the position of the left-top corner and size of its bounding rectangle. If the last parameter is omitted, the current drawing color is used.
<code>void fillPolygon(/*List of coordinates*/)</code>	Renders a solid polygon with the current drawing color. The parameters are <code>(x,y)</code> pairs of integers specifying the polygon's vertices (corners).

Table 12.1: A subset of the drawing methods provided by the `Viewport` class, a rectangular drawing surface. These methods **should not be used outside** of the `draw()` method.

More complex pictures can be drawn by combining several primitive shapes. These drawing methods **should not be used outside** of the `draw()` method. Furthermore, code you write **should not attempt to call** the `draw()` method directly. It is the responsibility of the window manager to call your viewport's `draw()` method.²

The `Viewport` class defines a number of public constant color objects that affect drawing. These colors include `RED`, `BLUE`, `GREEN`, `YELLOW`, `CYAN`, `MAGENTA`, `GRAY`, `ORANGE`, `PINK`, `LIGHT_GRAY`, `DARK_GRAY`, `WHITE`, `BLACK`, and `TRANSPARENT`. These are `java.awt.Color` objects. If the predefined colors do not meet your need, you can make your own color object by specifying the proper combination of red-green-blue (RGB) primary color values. The primary color values are integers that can range from 0 to 255. Black is `(0,0,0)` representing no contribution by any of the primary colors; white is `(255,255,255)` representing all primary colors contributing fully. The following statement

```
Color lightGreen = new Color(100, 255, 100);
```

creates a color object with red contributing $\frac{100}{255}$, green contributing fully ($\frac{255}{255}$), and contributing $\frac{100}{255}$. This results in a lighter shade of green than `Viewport.GREEN` which has an RGB combination of `(0,255,0)`.

The background color of a viewport can be set with `setBackground()` that accepts a single `Color` parameter.

To see how this all works, let us draw a static picture of a traffic light. Our simple graphical light will be composed of a rectangular gray frame and three circular lamps of the appropriate colors. Figure 12.2 shows our

²Actually the window manager calls the viewport's `repaint()` method which through a chain of standard graphical painting methods eventually calls the viewport's `draw()` method.

graphical design. The numbers shown are pixel values. The rectangular frame can be rendered with the viewport's

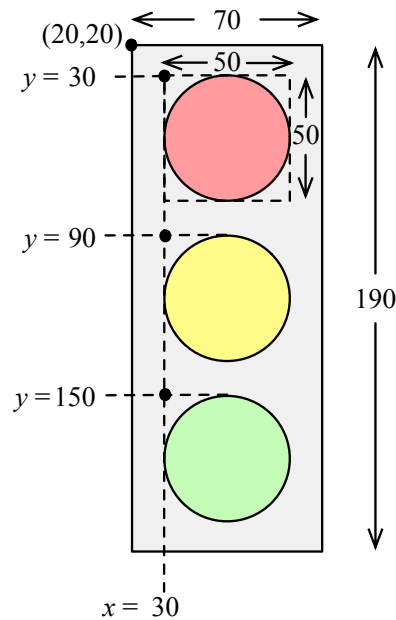


Figure 12.2: Graphical traffic light design.

`fillRectangle()` method. The `fillRectangle()` method expects an (x,y) value for the rectangle's left-top corner, a width value, a height value, and a color. According to the figure we would use the call

```
fillRectangle(20, 20, 70, 190, GRAY);
```

Each lamp is an ellipse with a square bounding box (width = height, effectively drawing a circle). The top (red) lamp's bounding box is offset 10 pixels horizontally and vertically from the frame's left-top corner, and its diameter is 50 pixels. The following statement is exactly what we need:

```
fillOval(30, 30, 50, 50, RED);
```

The remaining lamps are the same size (50×50) and offset the same vertically (30 pixels). They obviously need bigger vertical offsets. If we want 10 pixels between each lamp, and the lamps are 50 pixels across, we need to add 60 to the y value of the previous lamp's drawing statement. The statements

```
fillOval(30, 90, 50, 50, YELLOW);
fillOval(30, 150, 50, 50, GREEN);
```

will work nicely.

The four drawing statements we derived above must be placed in the `draw()` method of a `Viewport` subclass. `DrawTrafficLight` (Figure 12.1) works nicely:

```
import edu.southern.computing.oopj.Viewport;

public class DrawTrafficLight extends Viewport {
    public DrawTrafficLight() {
        super("Traffic Light", 100, 100, 50, 270);
    }
}
```



```

    }
    public void draw() {
        fillRectangle(20, 20, 70, 190, GRAY);
        fillOval(30, 30, 50, 50, RED);
        fillOval(30, 90, 50, 50, YELLOW);
        fillOval(30, 150, 50, 50, GREEN);
    }
}

```

Listing 12.1: DrawTrafficLight—Renders a graphical traffic light

The following interactive session draws the light:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> new DrawTrafficLight();

```

DrawStar (Figure 12.2) uses the drawPolygon() method to draw a five-pointed star:

```

import edu.southern.computing.oopj.Viewport;

public class DrawStar extends Viewport {

    public DrawStar() {
        super("Star", 100, 100, 200, 200);
        setBackground(WHITE);
    }

    public void drawStar(int xCenter, int yCenter) {
        // Code adapted from http://www.research.att.com/
        //                      sw/tools/yoix/doc/graphics/
        //                      pointInPolygon.html
        drawPolygon(xCenter + 0, yCenter - 50,
                    xCenter + 29, yCenter + 40,
                    xCenter - 47, yCenter - 15,
                    xCenter + 47, yCenter - 15,
                    xCenter - 29, yCenter + 40);
    }

    public void draw() {
        drawStar(getWidth()/2, getHeight()/2);
    }

    public static void main(String[] args) {
        new DrawStar();
    }
}

```

Listing 12.2: DrawStar—draws a five-pointed star

As in the case of DrawTrafficLight (Figure 12.1), all drawing is performed within the overridden draw() method.

12.3 Event Model

Users interact differently with graphical programs than they do with text-based programs. A text-based program predetermines when input is needed. For example, suppose scan is a Scanner object. To receive an integer value from the user the statement

```
int value = scan.nextInt();
```

causes the program's execution to come to a complete halt until the user provides the requested value.

In a graphical program, the user may move the mouse pointer over particular portion of the window and then click and drag the mouse. Instead, the user may select an item from the menu bar, or right click the mouse button to bring up a context-sensitive popup menu. Perhaps user presses a keyboard shortcut key like **Ctrl S**. Each of these activities, such as moving the mouse, pressing the mouse button, typing a key, or selecting a menu item, is called an *event*. The window manager monitors the program's execution watching for events to transpire. When an event occurs, the window manager notifies the running program. The program in turn either responds to or ignores the event.

Our viewport objects can respond to mouse and key events. The window manager notifies our viewport object that an event has occurred by calling a method that corresponds to that event. Viewport objects currently can handle the events shown in Table 12.2. By default, these methods do nothing. In order to allow your viewport to respond

Event	Method
Mouse button pressed	mousePressed()
Mouse button released	mouseReleased()
Mouse button clicked	mouseClicked()
Mouse pointer moved over viewport from outside	mouseEntered()
Mouse pointer moved out of viewport	mouseExited()
Mouse moved while button depressed	mouseDragged()
Mouse moved with no buttons depressed	mouseMoved()
Key typed	keyTyped()

Table 12.2: Viewport events and their corresponding methods

to events in ways appropriate for your application, subclass Viewport and override the corresponding methods. SimpleResponder (Figure 12.3) illustrates how this is done:

```
import edu.southern.computing.oopj.Viewport;

public class SimpleResponder extends Viewport {
    public SimpleResponder() {
        super("Very Simple Responder", 100, 100, 400, 300);
    }
}
```

```

    }
    // The window manager calls this method when the user depresses
    // the mouse button when the mouse pointer is over the viewport
    public void mousePressed() {
        System.out.println("Mouse is at (" + getMouseX() + ", "
                           + getMouseY() + ")");
    }
    // The window manager calls this method when the user types a
    // a key when the viewport has the focus
    public void keyTyped() {
        System.out.println("'" + getKeyTyped() + "' typed");
    }
}

```

Listing 12.3: SimpleResponder—monitors two kinds of events—mouse button presses and keystrokes

In the Interactions pane simply create an instance of SimpleResponder. See what happens when you click the mouse and press keys in the viewport.

If your custom viewport class does not override an event method, it in effect ignores that event. In reality the window manager calls the appropriate method anyway, but the empty body does nothing. It is common for applications not to respond to various events, so often your custom viewport will override few, if any, of the event methods.

SimpleResponder (▢ 12.3) reveals several other viewport methods that are valuable when writing event handlers:

- `getMouseX()`—returns the *x* coordinate of the mouse pointer's location when the mouse event occurred,
- `getMouseY()`—returns the *y* coordinate of the mouse pointer's location when the mouse event occurred, and
- `getKeyTyped()`—returns the character corresponding to the key typed.

These methods need not be used at all; sometimes it is sufficient to know that the mouse button was pressed, and your application does not care where the mouse was at the time. Similarly, sometimes your application needs to know when any key is typed and does not care about which particular key was typed. InteractiveTrafficLightViewport (▢ 12.4) shows how the `mouseClicked()` method can be overridden using neither `getMouseX()` nor `getMouseY()`. It makes use of `TrafficLightModel` (▢ 10.1).

```

import edu.southern.computing.oopj.Viewport;

public class InteractiveTrafficLightViewport extends Viewport {
    // The traffic light model controls the state of the light
    private TrafficLightModel model;

    public InteractiveTrafficLightViewport() {
        super("Traffic Light---Click to change", 100, 100, 50, 270);
        model = new TrafficLightModel(TrafficLightModel.STOP);
    }
}

```

```

//  Conditionally illuminate lamps based on the state of the
//  traffic light model
public void draw() {
    fillRectangle(20, 20, 70, 190, GRAY);
    int state = model.getState();
    if (state == TrafficLightModel.STOP) {
        fillOval(30, 30, 50, 50, RED);
    } else {
        fillOval(30, 30, 50, 50, BLACK);
    }
    if (state == TrafficLightModel.CAUTION) {
        fillOval(30, 90, 50, 50, YELLOW);
    } else {
        fillOval(30, 90, 50, 50, BLACK);
    }
    if (state == TrafficLightModel.GO) {
        fillOval(30, 150, 50, 50, GREEN);
    } else {
        fillOval(30, 150, 50, 50, BLACK);
    }
}

//  The window manager calls this method when the user clicks
//  the mouse button when the mouse pointer is over the viewport
public void mouseClicked() {
    model.change();
}
}

```

Listing 12.4: InteractiveTrafficLightViewport—a simple interactive graphical traffic light

In InteractiveTrafficLightViewport (▮12.4), the user simply clicks the mouse over the window to change the traffic light.

12.4 Anonymous Inner Classes

In Java, a class can be defined within another class's definition. The enclosed class is called an *inner class*, and the enclosing class is called the *outer class*. While it is legal and sometimes beneficial to do so, we often will not need to define such named nested classes. One related aspect of nested classes is extremely useful, however. Sometimes it is convenient to create an instance of a subclass without going to the trouble of defining the separate, named subclass. For example, recall SimpleResponder (▮12.3). Clearly SimpleResponder is a subclass of Viewport, and this subclassing is necessary so we can override various methods to achieve the interactive viewport we want. We can write a main program that uses SimpleResponder (▮12.3) as shown in UsingSimpleResponder (▮12.5):

```
import edu.southern.computing.oopj.Viewport;
```

```
public class UsingSimpleResponder {
    public static void main(String[] args) {
        new SimpleResponder();
    }
}
```

Listing 12.5: UsingSimpleResponder—uses the SimpleResponder class.

SimpleResponder is a simple straightforward extension of the Viewport class. If we never need to use the SimpleResponder class ever again, it seems wasteful to go to the trouble to define a separate class which requires the creation of an associated SimpleResponder.java source file. Observe that a SimpleResponder object is a straightforward extension of a plain Viewport object, as only two methods are overridden. Ideally, we should be able to create a Viewport object that has a little added functionality over the stock viewport without going to the trouble of defining a new named class. Java's *anonymous inner class* feature allows us to do so.

Creating an anonymous inner class is easy. It looks like a combination of object creation and class definition, and indeed that is exactly what it is. AnonymousSimpleResponder (Listing 12.6) avoids defining a separate named subclass of Viewport:

```
import edu.southern.computing.oopj.Viewport;

public class AnonymousSimpleResponder {
    public static void main(String[] args) {
        new Viewport("Anonymous Simple Responder",
                     100, 100, 400, 300) {
            // What to do when the mouse is pressed
            public void mousePressed() {
                System.out.println("Mouse is at (" + getMouseX() + ", "
                                   + getMouseY() + ")");
            }
            // The window manager calls this method when the user types a
            // a key when the viewport has the focus
            public void keyTyped() {
                System.out.println("'" + getKeyTyped() + "' typed");
            }
        };
    }
}
```

Listing 12.6: AnonymousSimpleResponder—avoids the use of the SimpleResponder class.

It initially appears that we are creating a simple Viewport object with the new operator, but the statement does not end there; in fact, this statement does not end until the semicolon after the close curly brace on the third-to-the-last line. What comes in between looks like the body of a class definition, and indeed it is. It is the definition of

the anonymous inner class. One source file is created (`AnonymousSimpleResponder.java`), but the compiler creates two bytecode files: `AnonymousSimpleResponder.class` and `AnonymousSimpleResponder$1.class`. The second file contains the anonymous inner class code.

The nice thing about inner classes is they have access to the methods and variables of their enclosing outer classes. The next section shows how this capability is very useful.

12.5 A Popup Menu Class

Another convenience class, `ContextMenu` enables programmers to add a popup menu to a viewport. `ContextMenu` objects are very simple—the class has one constructor and one method. The constructor accepts any number of strings representing menu items. The `handler()` method accepts a single string parameter. A viewport's popup menu is set via `setContextMenu()`. When the user clicks the right mouse button (in Mac OS X, **Ctrl** click triggers the popup menu) the popup menu appears. If the user selects an item from the menu, the event manager calls the `handler()` method of the menu, passing the name of the selected menu item. A conditional statement within `handler()` decides what action should be taken.

`SimpleMenuExample` (Figure 12.7) shows how the process works:

```
import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.ContextMenu;

public class SimpleMenuExample {
    private static int value = 0;
    public static void main(String[] args) {
        Viewport window = new Viewport("Useless Default Menu",
                                         100, 100, 400, 300) {
            public void draw() {
                drawString(Integer.toString(value), 50, 80);
                if (value > 0) {
                    fillRectangle(50, 120, 5*value, 20,
                                Viewport.BLUE);
                } else {
                    fillRectangle(50 + 5*value, 120, -5*value,
                                20, Viewport.RED);
                }
            }
        };

        window.setContextMenu(new ContextMenu("Increase",
                                              "Decrease",
                                              "Quit") {
            public void handler(String item) {
                if (item.equals("Increase")) {
                    value++;
                } else if (item.equals("Decrease")) {
                    value--;
                } else if (item.equals("Quit")) {
                    System.exit(0);
                }
            }
        });
    }
}
```

```

        }
    });
}
}

```

Listing 12.7: SimpleMenuExample—uses a popup menu to modify the displayed number.

SimpleMenuExample (Figure 12.7) displays two pieces of information:

- a number representing the current value of the `value` variable and
- a bar with a length that reflects `value`'s value. (What happens when you decrease `value` to below zero?)

Study the structure of SimpleMenuExample (Figure 12.7) very carefully. It uses two anonymous inner classes:

- The `window` variable is assigned an instance of a class derived from `Viewport`.
- The parameter of the invocation of `setContextMenu()` is an instance of a subclass of `ContextMenu`.

An anonymous inner class can define instance variables just like any other class. Anonymous inner classes cannot have constructors, since by definition a constructor has the same name as the class, but the class has no name!

12.6 Summary

The `edu.southern.computing.oopj.Viewport` class is a subclass of the standard class `javax.swing.JPanel`. The Sun Java documentation and many books cover the `JPanel` class well. The following lists some useful methods in the `Viewport` class and also includes `JPanel` methods that are commonly used in simple graphics programs:

Constructor

- `Viewport(String title, int x, int y, int width, int height)`
Creates a new `width×height` viewport object with `title` in its titlebar, and left-top corner at (x,y) .

Manipulation

- `int getX()`
Returns the x coordinate of the viewport's left-top corner.
- `int getY()`
Returns the y coordinate of the viewport's left-top corner.
- `int getWidth()`
Returns the width of the viewport in pixels.
- `int getHeight()`
Returns the height of the viewport in pixels.

- `int setLocation(int x, int y)`
Sets the left-top corner of the viewport to location (x,y) . The viewport's size is unchanged.
- `int setSize(int w, int h)`
Sets the width and height of the viewport to $w \times h$. The viewport's left-top corner location is unchanged.

Mouse Events

- `void mousePressed()`
Called by the window manager when the user presses the left mouse button when the mouse pointer is within the viewport.
- `void mouseReleased()`
Called by the window manager when the user releases the left mouse button when the mouse pointer is within the viewport.
- `void mouseClicked()`
Called by the window manager when the user presses and releases the left mouse button when the mouse pointer is within the viewport.
- `void mouseEntered()`
Called by the window manager when the user moves the mouse cursor into the viewport from outside the viewport.
- `void mouseExited()`
Called by the window manager when the user moves the mouse cursor out of the viewport from inside the viewport.
- `void mouseMoved()`
Called by the window manager when the user moves the mouse cursor within the viewport while no mouse buttons are depressed.
- `void mouseDragged()`
Called by the window manager when the user moves the mouse cursor within the viewport while a mouse button is depressed.
- `int getMouseX()`
Returns the x coordinate of the mouse cursor during the previous mouse event.
- `int getMouseY()`
Returns the y coordinate of the mouse cursor during the previous mouse event.

Keyboard Events

- `void keyTyped()`
Called by the window manager when the user types a key when the window has keyboard focus.
- `char getKeyTyped()`
Returns the character typed during the previous keyboard event.

Graphics

- `void draw()`

Called by the window manager when the contents of the viewport needs to be displayed. All of the following methods are ordinarily called by programmer-written code within the `draw()` method.

- `void setColor(Color color)`

Sets the current drawing color to `color`. Several predefined constants are available: `BLACK`, `WHITE`, `RED`, `BLUE`, `GREEN`, `YELLOW`, `CYAN`, `GRAY`, `DARK_GRAY`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, and `TRANSPARENT`.

- `void drawPoint(int x, int y, Color color)`

An overloaded method; parameter `color` is optional. Draws a single pixel point at (x,y) with color `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void drawLine(int x1, int y1, int x2, int y2, Color color)`

An overloaded method; parameter `color` is optional. Draws a line connecting points $(x1,y1)$ and $(x2,y2)$ with color `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void drawRectangle(int x, int y, int width, int height, Color color)`

An overloaded method; parameter `color` is optional. Draws a `width`×`height` rectangle with left-top corner at (x,y) . The rectangle's color is `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void fillRectangle(int x, int y, int width, int height, Color color)`

An overloaded method; parameter `color` is optional. Draws a `width`×`height` rectangle with left-top corner at (x,y) . The interior of the rectangle is filled completely with the color specified by `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void drawOval(int x, int y, int width, int height, Color color)`

An overloaded method; parameter `color` is optional. Draws an ellipse totally contained by a `width`×`height` bounding rectangle with left-top corner at (x,y) . The ellipse's color is `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void fillOval(int x, int y, int width, int height, Color color)`

An overloaded method; parameter `color` is optional. Draws an ellipse totally contained by a `width`×`height` bounding rectangle with left-top corner at (x,y) . The ellipse's interior is filled completely with the color specified by `color`. If the `color` parameter is omitted, the current drawing color is used.

- `void drawPolygon(list of integers)`

Draws a polygon with vertices at the locations specified in the parameter list. The first integer is the x coordinate of the first vertex, the second parameter is the y coordinate of the first vertex, the third parameter is the x coordinate of the second vertex, the fourth parameter is the y coordinate of the second vertex, etc. The current drawing color is used.

- `void fillPolygon(list of integers)`

Works like `drawPolygon()`, but a filled polygon is drawn instead of an outline. The current drawing color is used.

- `void drawString(String message, int x, int y)`

Draws a string at location (x,y) . The current drawing color is used.

12.7 Exercises

1. Make a new interactive graphical traffic light that uses `TurnLightModel` (Figure 11.1) as its model. Your new class should properly display a green turn arrow.

Chapter 13

Some Standard Java Classes

Java provides a plethora of predefined classes that programmers can use to build sophisticated programs. If you have a choice of using a predefined standard class or devising your own custom class that provides the same functionality, choose the standard class. The advantages of using a standard class include:

- The effort to produce the class is eliminated entirely; you can devote more effort to other parts of the application's development.
- If you create your own custom class, you must thoroughly test it to ensure its correctness; a standard class, while not immune to bugs, generally has been subjected to a complete test suite. Standard classes are used by many developers and thus any lurking errors are usually exposed early; your classes are exercised only by your code, and errors may not be manifested immediately.
- Other programmers may need to modify your code to fix errors or extend its functionality. Standard classes are well known and trusted; custom classes, due to their limited exposure, are suspect until they have been thoroughly examined. Standard classes provide well-documented, well-known interfaces; the interfaces of custom classes may be neither.

Software development today is increasingly *component-based*. In the hardware arena, a personal computer is built by assembling

- a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips),
- a processor and its various support chips,
- memory boards,
- a video card,
- an input/output card (serial ports, parallel port, mouse port)
- a disk controller,
- a disk drive,
- a case,
- a keyboard,

- a mouse, and
- a monitor.

The video card is itself a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the OS).

Software components are used like hardware components. A software system can be built largely by assembling pre-existing software building blocks. A class is a simple software component; more sophisticated software components can consist of collections of collaborating classes. Java's standard classes are analogous to off-the-shelf hardware components.

13.1 Packages

Java includes a multitude of classes. These classes are organized into collections called *packages*. In order to speed compilation and class loading during execution, the compiler normally has access to a small subset of the standard classes. These common classes include `String` and `System` (as in `System.out.println()`) and are located in a package called `java.lang`. Other classes, some of which we consider in this chapter, require a special statement at the top of the source code that uses these classes:

```
import java.util.Random;
```

This `import` statement makes the definition of the `Random` class available to the compiler. The `Random` class is used to make random number generators which are useful for games and some scientific applications. The `Random` class is explored in § 13.6. The `Random` class is part of the `java.util` package of classes that contain utility classes useful for a variety of programming tasks. Access to the definition of `Random` enables the compiler to verify that client code uses `Random` objects properly. Without the `import` statement that compiler will flag as errors any statements that use `Random`.

`Random`, `String`, and `System` are *simple* class names. The full class name includes the package in which the class is a member. The full class names for the above classes are `java.util.Random`, `java.lang.String`, and `java.lang.System`. If full names are used all the time, no `import` statements are required; however, most programmers will import the classes so the shorter simple names can be used within a program. Full names are required when two classes with the same name from different packages are to be used simultaneously.

Multiple `import` statements may be used to import as many class definitions as required. If present, these `import` statements must appear before any other lines in the source code, except for comments.

13.2 Class System

We have seen the `print()`, `printf()`, and `println()` methods provided by the `out` public class constant of the `System` class. The `System` class also provides a number of other services, one of which is a class method named `currentTimeMillis()`. The method `System.currentTimeMillis()` gets information from the operating system's clock to determine the number of milliseconds since midnight January 1, 1970. By calling the method twice and comparing the two results we can determine the time elapsed between two events. Try the following interactive session:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> long start = System.currentTimeMillis();
> System.currentTimeMillis() - start
30166
```

Type in the first statement and, just before hitting **Enter**, note the time on a clock. Go ahead and type the second line but wait to hit the **Enter** key until about 30 seconds has elapsed since last pressing the **Enter** key (if you are a slow typist, you may want to wait a full minute). The value displayed is the number of milliseconds ($\frac{1}{1000}$ seconds) between the two **Enter** presses.

Using `System.currentTimeMillis()` interactively to time human user activity is rather limited, but `System.currentTimeMillis()` can be very useful to time program execution and synchronizing program activity with particular time periods. Stopwatch (Figure 13.1) implements a software stopwatch that we can use to time program execution:

```
public class Stopwatch {
    // A snapshot of the system time when the stopwatch is started
    private long startTime;
    // The number of milliseconds elapsed from when the stopwatch was
    // started to when it was stopped
    private long elapsedTime;
    // Is the stopwatch currently running?
    private boolean running = false;
    // Start the watch running
    public void start() {
        running = true;
        startTime = System.currentTimeMillis();
    }
    // Stop the watch and note elapsed time
    public void stop() {
        elapsedTime = System.currentTimeMillis() - startTime;
        running = false;
    }
    // Return the elapsed time
    public long elapsed() {
        // This method should only be called on a stopped watch
        if ( running ) { // Issue warning message
            System.out.println("**** Stopwatch must be stopped to "
                               + "provide correct elapsed time.");
        }
        return elapsedTime;
    }
}
```

Listing 13.1: Stopwatch—implements a software stopwatch to time code execution

To use Stopwatch objects:

1. Create a stopwatch object: `Stopwatch s = new Stopwatch();`
2. At the beginning of an event start it: `s.start();`
3. At the end of the event stop it: `s.stop();`
4. Read the elapsed time: `s.elapsed();`

13.3 Class String

We have been using strings so much they may seem like primitive types, but strings are actually `String` objects. A string is a sequence of characters. In Java, a `String` object encapsulates a sequence of Unicode (see <http://www.unicode.org>) characters and provides a number of methods that allow programmers to use strings effectively. Since `String` is a member of the `java.lang` package, no import is necessary.

A *string literal* is a sequence of characters enclosed within quotation marks, as in

```
"abcdef"
```

A string literal is an instance of the `String` class. The statement

```
String w = "abcdef";
```

assigns the `String` object reference variable `w` to the `String` object `"abcdef"`. To make a copy of this literal string, the `new` operator must be used:

```
String w = new String("abcdef");
```

This assignment directs `w` to refer to a distinct object from the literal `"abcdef"` object, but the characters that make up the two strings will be identical.

Table 13.1 lists some of the more commonly used methods of the `String` class.

Some Methods of the String Class	
char charAt(int i)	Returns the character at the specified index i. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on.
int length()	Returns the number of characters in a string
boolean equals(Object s)	Determines if one string contains exactly the same characters in the same sequence as another string
String concat(String str)	Splices one string onto the end of another string
int indexOf(int c)	Returns the position of the first occurrence of a given character within a string
String toUpperCase()	Returns a new string containing the capitalized version of the characters in a given string. The string used to invoke toUpperCase() is not modified.
String valueOf(int i)	Converts an int into its string representation. The method is overloaded for all the primitive types.

Table 13.1: A subset of the methods provided by the String class

StringExample (Figure 13.2) exercises some string methods.

```
public class StringExample {
    public static void test() {
        String str = "abcdef";
        // Should print 6
        System.out.println(str.length());
        // Also should print 6
        System.out.println("abcdef".length());
        // Use the special concatenation operator
        System.out.println(str + "ghijk");
        // Use the concatenation method
        System.out.println(str.concat("ghijk"));
        // See the uppercase version of the string
        System.out.println(str.toUpperCase());
        // Should print "true"
        System.out.println(str.equals("abcdef"));
        // Should print "[c]"
        System.out.println "[" + str.charAt(2) + "]";
        // Should print 2
        System.out.println("Character c is in position " + str.indexOf('c'));
    }
}
```

Listing 13.2: StringExample—Illustrates some string methods

Some parts of `StringExample` are worth special attention:

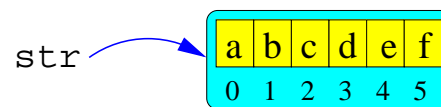
- The statement

```
System.out.println("abcdef".length());
```

emphasizes the fact that a string literal in Java is an actual object.

- The `+` operator and `concat()` method are two different ways to concatenate strings in Java. The `String` class is special in Java because no other classes can use an operator to invoke a method. The `+` operator executes the `concat()` method.
- The first character in a string is at position zero, the second character is at position one, and so forth. Figure 13.1 illustrates.

```
String str = "abcdef";
```



```
str.charAt(2) == 'c'
```

Figure 13.1: String indexing. The character 'c' is found at index 2.

- Observe that none of the methods used in `StringExample` modify the contents of `str`. The expression

```
str.toUpperCase()
```

returns a new string object with all capital letters; it does *not* modify `str`. In fact, none of the methods in the `String` class modify the string object upon which they are invoked. A string object may *not* be modified after it is created. Such objects are called *immutable*. It is acceptable to modify a reference, as in:

```
str = str.toUpperCase();
```

This statement simply reassigns `str` to refer to the new object returned by the method call. It does not modify the string object to which `str` was originally referring. Figure 13.2 illustrates.

One additional useful `String` method is `format()`. It is a class (static) method that uses a format string like `System.out.printf()`. Whereas `System.out.printf()` displays formatted output, `String.format()` instead produces a string, as demonstrated in the following interaction:

Interactions

```
Welcome to DrJava. Working directory is /Users/rick/java
> double x = 3.69734;
> x
3.69734
> String.format("%.2f", x)
"3.70"
```

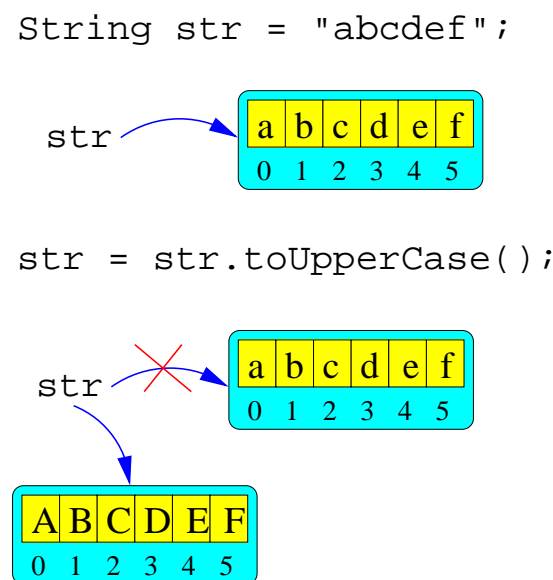



Figure 13.2: String reassignment. The reference is redirected to point to a different string object; the original string object is unaffected (but it may be garbage collected if it is no longer referenced elsewhere).

The `%.2f` control code specifies a floating point number rounded to two decimal places. The control codes used in the `String.format()` format string are exactly those used the `System.out.printf()` format string. This is because the two methods use a common formatting object (of class `java.util.Formatter`) to do their work.

13.4 Primitive Type Wrapper Classes

Java's distinction between primitive types and reference types dilutes its object-orientation. Some OO languages, like Smalltalk, have no primitive types—everything is an object. As we'll discover, it is often convenient to treat a primitive type as if it were an object.

The `java.lang` package provides a “wrapper” class for each primitive type. These wrapper classes serve two roles:

1. They can be used to wrap a primitive type value within an object. This objectifies the primitive value and adds a number of useful methods that can act on that value. The state of the object is defined solely by the value of the primitive type that it wraps.
2. They provide services or information about the primitive type they wrap. For example:
 - The minimum and maximum values that the primitive type can represent are available.
 - Conversions to and from the primitive type's human-readable string representations can be performed.

Table 13.2 lists all of Java's wrapper classes.

We'll focus on the `Integer` class, but the other wrapper classes provide similar functionality for the types they wrap. Some useful `Integer` methods are shown in Table 13.3.

Wrapper Class	Primitive Type
Byte	byte
Short	short
Character	char
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean

Table 13.2: Wrapper classes used to objectify primitive types

Some Methods of the Integer Class
<pre>static int parseInt(String s)</pre> <p>A class method that converts a string representation of an integer to an <code>int</code>. For example,</p> <pre>int value = Integer.parseInt("256");</pre> <p>Note that the following statement is illegal:</p> <pre>int value = "256"; // Illegal!</pre> <p>since a <code>String</code> object reference is not assignment compatible with an <code>int</code>.</p>
<pre>static int parseInt(String s, int b)</pre> <p>This overloaded method works with different number bases. For example,</p> <pre>int value = Integer.parseInt("101", 2);</pre> <p>assigns the value five to <code>value</code>, since $101_2 = 5_{10}$.</p>
<pre>static Integer valueOf(String s)</pre> <p>Like <code>parseInt()</code>, but returns a reference to a new <code>Integer</code> object, not a primitive <code>int</code> value.</p>
<pre>static Integer valueOf(String s, int b)</pre> <p>As with <code>parseInt()</code>, an overloaded version that deals with different number bases.</p>
<pre>int intValue()</pre> <p>Returns the primitive <code>int</code> value wrapped by this <code>Integer</code> object. Similar methods exist for the other primitive numeric types.</p>
<pre>String toString()</pre> <p>Returns the <code>String</code> representation the wrapped <code>int</code> value.</p>
<pre>static String toString(int i)</pre> <p>A class method that returns the <code>String</code> representation the value of <code>i</code></p>

Table 13.3: Some useful Integer methods

Public constant fields `MIN_VALUE` and `MAX_VALUE` store, respectively, the smallest and largest values that can be represented by the `int` type. The `Integer` constructor is overloaded and accepts either a single `int` or single `String` reference as a parameter. `IntegerTest` (▮ 13.3) shows how the `Integer` class can be used.

```
public class IntegerTest {
```

```

public static void main(String[] args) {
    Integer i1 = new Integer("500"); // Make it from a string
    Integer i2 = new Integer(200);   // Make it from an int
    int i;

    i = i1.intValue();
    System.out.println("i = " + i); // Prints 500
    i = i2.intValue();
    System.out.println("i = " + i); // Prints 200
    i = Integer.parseInt("100");
    System.out.println("i = " + i); // Prints 100
    i1 = Integer.valueOf("600");
    i = i1.intValue();
    System.out.println("i = " + i); // Prints 600
    i = Integer.valueOf("700").intValue();
    System.out.println("i = " + i); // Prints 700
}
}

```

Listing 13.3: IntegerTest—Exercises the Integer class

A technique called *autoboxing* allows a primitive type to be converted automatically to an object of its wrapper type (*boxing*), and it allows an object of a wrapper type to be converted automatically to its corresponding primitive value (*unboxing*). For example, the compiler converts the following statement

```
Double d = 3.14; // 3.14 is boxed
```

into

```
Double d = new Double(3.14); // How the compiler treats it
```

Similarly, the following code converts the other direction:

```
Double d1 = new Double(10.2);
double d2 = d1; // 10.2 is unboxed
```

Boxing and unboxing is performed automatically during parameter passing and method returns as well. A method that expects a wrapper type object will accept a compatible primitive type and vice-versa. Autoboxing largely eliminates the need to use explicit constructor calls and methods such as `intValue()`. These methods are present because earlier versions of Java did not support autoboxing.

13.5 Class Math

The `Math` class contains mathematical methods that and provide much of the functionality of a scientific calculator. Table 13.4 lists only a few of the available methods.

Some Methods of the Math Class	
static double sqrt(double x)	Computes the square root of a number: $\sqrt{x} = \text{sqrt}(x)$.
static double pow(double x, double y)	Raises one number to the power of another: $x^y = \text{pow}(x, y)$.
static double log(double x)	Computes the natural logarithm of a number: $\log_e x = \ln x = \log(x)$.
static double abs(double x)	Returns the absolute value of a number: $ x = \text{abs}(x)$.
static double cos(double x)	Computes the cosine of a value specified in radians: $\cos \frac{\pi}{2} = \cos(\text{PI}/2)$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent.

Table 13.4: Some useful Math methods

All the methods in the `Math` class are *class* methods (*static*), so they can be invoked without the need to create a `Math` object. In fact, the `Math` class defines a private constructor, so it is impossible to create a `Math` instance (§ 16.2). Two public class constants are also available:

- `PI`—the double value approximating π
- `E`—the double value approximating e , the base for natural logarithms.

Figure 13.3 shows a problem that can be solved using methods found in the `Math` class. A point (x,y) is to be rotated about the origin (point $(0,0)$), while the point (p_x,p_y) is fixed. The distance between the moving point and the fixed point at various intervals is to be displayed in a table.

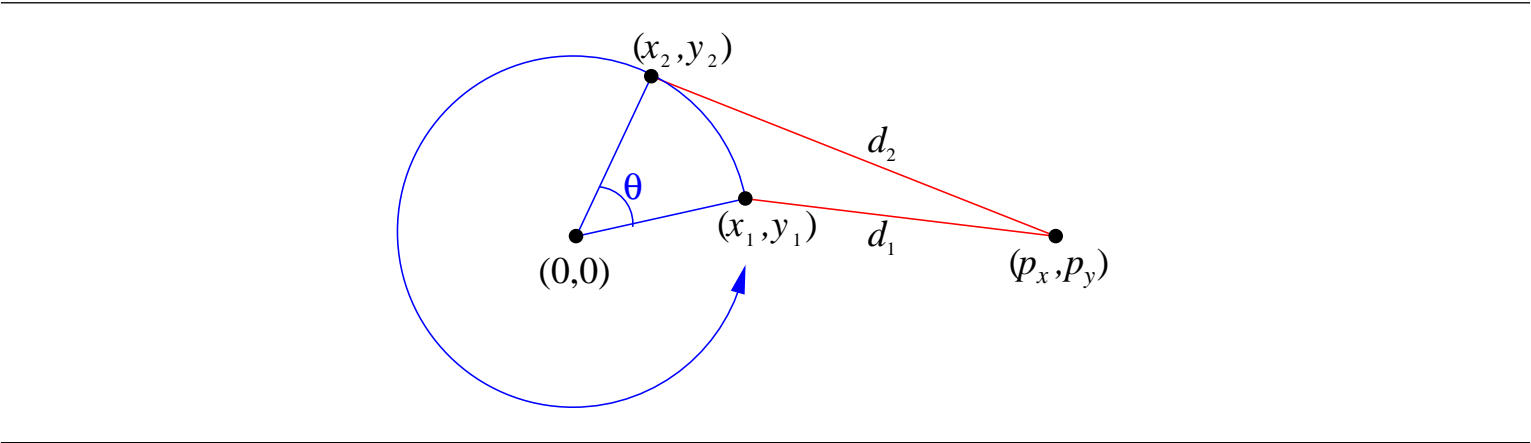


Figure 13.3: A problem that can be solved using methods in the `Math` class.

Point (p_x,p_y) could represent a spacecraft at a fixed location relative to a planet centered at the origin. The moving point could then represent a satellite orbiting the planet. The task is to compute the changing distance between the spacecraft and the satellite as the satellite orbits the planet. The spacecraft is located in the same plane as the satellite’s orbit.

Two problems must be solved, and facts from mathematics provide the answers:

1. **Problem:** The location of the moving point must be recomputed as it orbits the origin.

Solution: Given an initial position (x_1, y_1) of the moving point, a rotation of θ degrees around the origin will yield a new point at

$$\begin{aligned}x_2 &= x_1 \cos \theta - y_1 \sin \theta \\y_2 &= x_1 \sin \theta + y_1 \cos \theta\end{aligned}$$

2. **Problem:** The distance between the moving point and the fixed point must be recalculated as the moving point moves to a new position.

Solution: The distance d_1 between two points (p_x, p_y) and (x_1, y_1) is given by the formula

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

The distance d_2 in Figure 13.3 is

$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

OrbitDistanceTable (Figure 13.4) uses these mathematical results to compute a partial orbit distance table.

```
public class OrbitDistanceTable {
    // Location of orbiting point is (x,y)
    private static double x;
    private static double y;

    // Location of fixed point is always (100, 0), AKA (fixedX, fixedY)
    private static final double fixedX = 100;
    private static final double fixedY = 0;

    // Precompute the cosine and sine of 10 degrees
    private static final double COS10 = Math.cos(Math.toRadians(10));
    private static final double SIN10 = Math.sin(Math.toRadians(10));

    public static void move() {
        double xOld = x;
        x = x * COS10 - y * SIN10;
        y = xOld * SIN10 + y * COS10;
    }

    public static double distance(double x1, double y1, double x2, double y2) {
        return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
    }

    public static void report() {
        System.out.printf("%9.3f %9.3f %9.3f %n", x, y,
            distance(x, y, fixedX, fixedY));
    }

    public static void main(String[] args) {
        // Initial position of (x,y)
        x = 50;
        y = 0;
        System.out.println("      x      y      Distance");
    }
}
```

```

        System.out.println("-----");
        // Rotate 30 degrees in 10 degree increments
        report();
        move();    // 10 degrees
        report();
        move();    // 20 degrees
        report();
        move();    // 30 degrees
        report();
    }
}

```

Listing 13.4: OrbitDistanceTable—Generates a partial table of the orbital position and distance data

13.6 Class Random

Instances of the `java.util.Random` class are used to generate *random numbers*. Random numbers are useful in games and simulations. For example, many board games use a die (one of a pair of dice) to determine how many places a player is to advance. A software adaptation of such a game would need a way to simulate the random roll of a die.

All algorithmic random number generators actually produce *pseudorandom* numbers, not true random numbers. A pseudorandom number generator has a particular period, based on the nature of the algorithm used. If the generator is used long enough, the pattern of numbers produced repeats itself exactly. For example, a particularly poor pseudorandom number generator with a very short period may generate the sequence:

2013, 3, 138769, -2342, 7193787, 2013, 3, 138769, -2342, 7193787, ...

This repetition of identical values disqualifies all algorithmic generators from being true random number generators. The good news is that all practical algorithmic pseudorandom number generators have *much* longer periods.

Java's standard pseudorandom number generator is based on Donald Knuth's widely used linear congruential algorithm [3]. It works quite well for most programming applications requiring random numbers. The algorithm is given a seed value to begin, and a formula is used to produce the next value. The seed value determines the sequence of numbers generated; identical seed values generate identical sequences.

The `Random` class has an overloaded constructor:

- `Random()`—the no parameter version uses the result of `System.currentTimeMillis()` (the number of milliseconds since January 1, 1970) as the seed. Different executions of the program using a `Random` object created with this parameter will generate different sequences since the constructor is called at different times (unless the system clock is set back!).
- `Random(long seed)`—the single parameter version uses the supplied value as the seed. This is useful during development since identical sequences are generated between program runs. When changes are made to a malfunctioning program, reproducible results make debugging easier.

Some useful `Random` methods are shown in Table 13.5.

Some Methods of the Random Class	
<code>int nextInt(int n)</code> Overloaded: <ul style="list-style-type: none">• No parameter—returns the next uniformly distributed pseudorandom number in the generator’s sequence converted to an integer.• Single positive integer parameter (<i>n</i>)— works like the no parameter version except makes sure the result <i>r</i> is in the range $0 \leq r < n$.	
<code>double nextDouble()</code> Returns the next uniformly distributed pseudorandom number in the generator’s sequence converted to a double.	
<code>void setSeed(long seed)</code> Resets the seed value.	

Table 13.5: Some useful Random methods

To see how the Random class might be used, consider a *die*. A die is a cube containing spots on each of its six faces. The number of spots vary in number from one to six. The word *die* is the singular form of *dice*. Figure 13.4 shows a pair of dice.

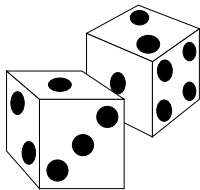


Figure 13.4: A pair of dice. The top face of the left die is one, and the top face of the right die is two.

Dice are often used to determine the number of moves on a board game and are used in other games of chance. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die.

DieGame (13.5) simulates rolling a pair of dice.

```
public class DieGame {
    public static void play() {
        Die die1 = new Die();
        Die die2 = new Die();
        // First, print the value on the faces
        System.out.println("Initially:");
        die1.show();
        die2.show();
        // Then, roll the dice
        die1.roll();
        die2.roll();
        // Finally, look at the results
        System.out.println("After rolling:");
    }
}
```

```

        die1.show();
        die2.show();
    }
}

```

Listing 13.5: DieGame—simulates rolling a pair of dice

A sample output of the DieGame would look like:

```

Initially:
+-----+
|  *   |
|      |
|      *  |
+-----+
+-----+
|  *   *  |
|      *  |
|  *   *  |
+-----+
After rolling:
+-----+
| * * * |
| * * * |
+-----+
+-----+
| *   *  |
|      *  |
|  *   *  |
+-----+

```

DieGame uses objects of the Die class (Figure 13.6). As DieGame shows, a Die object must:

- be created in a well-defined state, so `show()` can be invoked before `roll()` is called,
- respond to a `roll()` message to possibly change its visible face, and
- be able to show its top face in a text-based graphical fashion (rather than just printing the number of spots).

```

import java.util.Random;

public class Die {
    // The number of spots on the visible face
    private int value;
    // The class's pseudorandom number generator

```



```

private static Random random = new Random();
public Die() {
    roll();    // Initial value is random
}
// Sets value to a number in the range 1, 2, 3, ..., 6
public void roll() {
    value = random.nextInt(6) + 1;
}
// Draws the visible face in text-based graphics
public void show() {
    System.out.println("+-----+");
    if (value == 1) {
        System.out.println("|           |");
        System.out.println("|      *      |");
        System.out.println("|           |");
    } else if (value == 2) {
        System.out.println("| *           |");
        System.out.println("|           |");
        System.out.println("|      *      |");
    } else if (value == 3) {
        System.out.println("|           * |");
        System.out.println("|      *      |");
        System.out.println("| *           |");
    } else if (value == 4) {
        System.out.println("| *      * |");
        System.out.println("|           |");
        System.out.println("| *      * |");
    } else if (value == 5) {
        System.out.println("| *      * |");
        System.out.println("|      *      |");
        System.out.println("| *      * |");
    } else if (value == 6) {
        System.out.println("| * * * |");
        System.out.println("|           |");
        System.out.println("| * * * |");
    } else { // Defensive catch all:
        System.out.println(" *** Error: illegal die value ***");
    }
    System.out.println("+-----+");
}
}

```

Listing 13.6: Die—simulates a die

The client code, `DieGame`, is shielded from the details of the `Random` class. A `Random` object is encapsulated within the `Die` object. Observe that within the `Die` class:

- The `value` field is an instance variable. Each `Die` object must maintain its own state, so `value` cannot be shared (that is, it cannot be a class variable).

- The `random` field is a class variable. The `random` variable references a `Random` object and is used to generate the pseudorandom numbers that represent the visible face after a roll. This object can be shared by all dice, since each call to its `nextInt()` method simply returns the next pseudorandom number in the sequence. The next value of the `value` field for a particular die is not and should not be influenced by the current state of that die. While it is not logically incorrect to make `random` an instance variable, it would be an inefficient use of time and space since
 1. Each time a new `Die` object is created, a new `Random` object must also be created. Object creation is a relatively time-consuming operation.
 2. Each `Random` object associated with each `Die` object occupies memory. If only one `Random` object is really required, creating extra ones wastes memory.
- The `random` field is initialized when declared, so it will be created when the JVM loads the class into memory.
- The constructor automatically initializes `value` to be a random number in the range $1, 2, 3, \dots, 6$ by calling the `roll()` method. Even though `roll()` contains only one statement that could be reproduced here, this approach avoids duplicating code. In terms of maintainability this approach is better, since if in the future the nature of a roll changes (perhaps a ten-sided die), code only needs to be modified in one place.

If no constructor were provided, `value` would by default be assigned to zero, not a valid value since all faces have at least one spot on them.

- The statement

```
value = random.nextInt(6) + 1;
```

assigns a number in the range $1, 2, \dots, 6$, since the expression `random.nextInt(6)` itself returns a number in the range $0, 1, \dots, 5$.

13.7 Summary

- Add summary items here.

13.8 Exercises

1. Devise a new type of traffic light that has `TrafficLightModel` (¶ 10.1) as its superclass but does something different from `TurnLightModel` (¶ 11.1). Test your new class in isolation.
2. Create a new intersection class (subclass of `Intersection` (¶ 10.4)) Test your new type of intersection.

Chapter 14

The Object Class

Chapter 13 examined a few of the hundreds of classes available in Java's standard library. One standard Java class that was not mentioned deserves special attention. It rarely is used directly by programmers, but all of Java's classes, whether standard or programmer-defined, depend on it for some of their basic functionality. This chapter is devoted to the `Object` class, the root of Java's class hierarchy.

14.1 Class Object

While a class may serve as the superclass for any number of subclasses, every class, except for one, has exactly one superclass. The standard Java class `Object` has no superclass. `Object` is at the root of the Java class hierarchy. When a class is defined without the `extends` reserved word, the class implicitly extends class `Object`. Thus, the following definition

```
public class Rational {
    /* Details omitted . . . */
}
```

is actually just a shorthand for

```
public class Rational extends Object {
    /* Details omitted . . . */
}
```

Since `Object` is such a fundamental class it is found in the `java.lang` package. Its fully qualified name, therefore, is `java.lang.Object`.

The *is a* relationship is *transitive* like many mathematical relations. In mathematics, for any real numbers x , y , and z , if $x < y$ and $y < z$, we know $x < z$. This is known as the transitive property of the less than relation. In Java, if class Z extends class Y , and class Y extends class X , we know a Z *is a* Y and a Y *is a* X . The *is a* relation is transitive, so we also know that a Z *is a* X . Since `Object` is at the top of Java's class hierarchy, because of the transitive nature of inheritance, any object, no matter what its type, *is a* `Object`. Therefore, any reference type can be passed to method that specifies `Object` as a formal parameter or be assigned to any variable with a declared type of `Object`.

14.2 Object Methods

Since every class except `Object` has `Object` as a superclass (either directly or indirectly), every class inherits the methods provided by the `Object` class. The `Object` class provides 11 methods, and two of the 11 `Object` methods are highlighted in Table 14.1.

Method Name	Purpose
<code>equals</code>	Determines if two objects are to be considered “equal”
<code>toString</code>	Returns the string representation for an object

Table 14.1: Two commonly used `Object` methods

The `equals()` method inherited from `Object` checks for *reference equality*; that is, two references are equal if and only if they refer to exactly the same object. This performs the same test as the `==` operator when it is applied to reference types. In general, `==` is not appropriate for comparing object references in most applications. Consider `EqualsTest` (Figure 14.1) which illustrates the differences between `equals()` and `==`.

```
public class EqualsTest {
    public static void main(String[] args) {
        String s1 = "hello",
               s2 = new String(s1), // s2 is new string, a copy of s1
               s3 = s1;             // s3 is an alias of s1
        System.out.println("s1 = " + s2 + ", s2 = " + s2 + ", s3 = " + s3);
        if ( s1 == s2 ) {
            System.out.println("s1 == s2");
        } else {
            System.out.println("s1 != s2");
        }
        if ( s1.equals(s2) ) {
            System.out.println("s1 equals s2");
        } else {
            System.out.println("s1 does not equal s2");
        }
        if ( s1 == s3 ) {
            System.out.println("s1 == s3");
        } else {
            System.out.println("s1 != s3");
        }
        if ( s1.equals(s3) ) {
            System.out.println("s1 equals s3");
        } else {
            System.out.println("s1 does not equal s3");
        }
        if ( s2 == s3 ) {
            System.out.println("s2 == s3");
        } else {
            System.out.println("s2 != s3");
        }
        if ( s2.equals(s3) ) {
```

```

        System.out.println("s2 equals s3");
    } else {
        System.out.println("s2 does not equal s3");
    }
}
}

```

Listing 14.1: EqualsTest—illustrates the difference between `equals()` and `==`

The output of EqualsTest is

```

s1 = hello, s2 = hello, s3 = hello
s1 != s2
s1 equals s2
s1 == s3
s1 equals s3
s2 != s3
s2 equals s3

```

The `equals()` method correctly determined that all three strings contain the same characters; the `==` operator correctly shows that `s1` and `s3` refer to exactly the same `String` object.

The `toString()` method is called when an object must be represented as a string. Passing an object reference to a `System.out.println()` call causes the compiler to generate code that calls the object's `toString()` method so “the object” can be printed on the screen. The `toString()` method inherited from `Object` does little more than return a string that includes the name of the object's class and a numeric code that is related to that object's address in memory. The numeric code is expressed in hexadecimal notation, base 16. Hexadecimal numbers are composed of the digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

(See § 4.1.) This numeric code is of little use to applications programmers except to verify that two references point to distinct objects. If the hexadecimal codes for two references differ, then the objects to which they refer must be different. Subclasses should override the `toString()` method to provide clients more useful information.

These two methods would be valuable additions to the `RationalNumber` class (▮7.5). The overridden `equals()` method would check to see if the numerator and denominator of another fraction, in reduced form, is equal to a given fraction, in reduced form. The overridden `toString()` method returns a string that humans can easily interpret as a fraction (for example, $\frac{1}{2}$ becomes "1/2". These methods could be written:

```

public boolean equals(Object r) {
    if ( r != null && r instanceof Rational ) {
        Rational thisRational = reduce(),
            thatRational = ((Rational) r).reduce();
        return thisRational.numerator == thatRational.numerator &&
            thisRational.denominator == thatRational.denominator;
    }
    return false;
}

```

```

    }
    public String toString() {
        return numerator + "/" + denominator;
    }

```

The `toString()` method is straightforward, but the `equals()` method requires some comment. The `equals()` method as defined in `Object` will accept a reference to *any* object, since any Java object *is a* `Object`. To properly override the method, the parameter must be declared to be of type `Object`. This means our method needs to ensure the actual parameter is of type `Rational`, and, if it is, it must be cast to type `Rational`. Another issue is that `null` is a literal value that can be assigned to any object reference, we must provide for the possibility that the client may try to pass a `null` reference. The client may do this either directly:

```

Rational frac = new Rational(1, 2);
if (frac.equals(null)) { /* Do whatever . . . */}

```

or indirectly:

```

Rational frac1 = new Rational(1, 2), frac2 = makeFract();
// The method makeFract() might return null, but the client
// may be oblivious to the fact.
if (frac1.equals(frac2)) { /* Do whatever . . . */}

```

Since the parameter's type is `Object`, the following code is legal:

```

Rational frac = new Rational(1, 2);
TrafficLightModel light = new TrafficLightModel(TrafficLightModel.RED);
if (frac.equals(light)) { /* Do whatever . . . */}

```

We probably never want a `Rational` object to be considered equal to a `TrafficLightModel` object, so we need to be sure our method would handle this situation gracefully. The `instanceof` operator returns `true` if a given object reference refers to an instance of the given class. The `instanceof` operator respects the *is a* relationship as `InstanceOfTest` (Figure 14.2) demonstrates.

```

class X {}

class Y extends X {}

class Z {}

public class InstanceOfTest {
    public static void main(String[] args) {
        Y y = new Y();           // Make a new Y object
        Object z = new Z();      // Make a new Z object
        if ( y instanceof Y ) {
            System.out.println("y is an instance of Y");
        } else {
            System.out.println("y is NOT an instance of Y");
        }
        if ( y instanceof X ) {

```

```

        System.out.println("y is an instance of X");
    } else {
        System.out.println("y is NOT an instance of X");
    }
    if ( z instanceof Y ) {
        System.out.println("z is an instance of Y");
    } else {
        System.out.println("z is NOT an instance of Y");
    }
}
}

```

Listing 14.2: InstanceOfTest—demonstrates the use of the instanceof operator

The output of InstanceOfTest is

```

y is an instance of Y
y is an instance of X
z is NOT an instance of Y
x is NOT an instance of Y

```

In InstanceOfTest, *y* is an instance of *Y* and also an instance of *X*; said another way, *y is a Y* and *y is a X*. On the other hand, *z* is not an instance of *Y*. Note that *z* was declared as an `Object` reference, not a `Z` reference. If *z*'s declared type is `Z`, then the compiler will generate an error for the expression `z instanceof Y` since it can deduce that *z* can never be an instance of *Y*. The `Object` type is assignment compatible with all other types so the expression `z instanceof Y` is not illegal when *z*'s declared type is a supertype of *Y*. Since the declared type of *z* and *x* are, respectively, `Object` and `X`, and both `Object` and `X` are supertypes of *Y*, all of the uses of the `instanceof` operator in InstanceOfTest are legal.

14.3 Summary

- `Object` in the class `java.lang` is the superclass of all Java classes, even programmer-defined classes.
- The *is a* relation is transitive, so any object reference *is a* `Object`.
- The `instanceof` reserved word is an operator used to determine if an object reference *is a* given type. The `instanceof` operator respects the transitivity of the *is a* relation.
- Of the 11 methods provided by `Object`, `equals()` and `toString()` are often overridden by subclasses.
- An object's `toString()` method is called when the compiler must produce a string representation of the object, such as for a call to `System.out.println()`.
- The `equals()` method's formal parameter in `Object`. Subclasses that override `equals()` must ensure that the actual parameter is non-null and has the exact type of the subclass itself.
- The programmer has complete control over what the overridden `toString()` and `equals()` methods do, but their behavior should agree with the natural expectations of the client.

14.4 Exercises

1. Modify the traffic light view classes, `TextTrafficLight` (▮10.2) and `TextTurnLight` (▮11.2) so that `toString()` replaces the `show()` methods. This will allow code originally written as

```
System.out.println(light.show());
```

to be written more simply as

```
System.out.println(light);
```

with the same behavior. Can you think of a way to make it work by modifying only the superclass, `TextTrafficLight`?

Chapter 15

Software Testing

We know that a clean compile does not imply that a program will work correctly. We can detect errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called *testing*. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. We will see that good testing requires the same skills and creativity as programming itself.

Until recently testing was often an afterthought. Testing was not seen to be as glamorous as designing and coding. Poor testing led to buggy programs that frustrated users. Also, tests were written largely after the program's design and coding were complete. The problem with this approach is major design flaws may not be revealed until late in the development cycle. Changes late in the development process are invariably more expensive and difficult to deal with than changes earlier in the process.

Weaknesses in the standard approach to testing led to a new strategy: *test-driven development*. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. Tests are developed before any application code is written, and any application code produced is immediately subjected to testing.

Object-oriented development readily lends itself to such a style of testing. When the nature of a class—its interface and functionality—has been specified, special client code can be written to create and exercise objects of that class. This client code cannot be executed until the class under test has been written, but it will be ready to go when then the class becomes available.

15.1 The `assert` Statement

The `assert` statement can be used to make a claim (an assertion) about some property within an executing program. Either the property is true, or it is false. If during the program's execution the property is true, the `assert` statement is as if it were not there; however, if the property is false, the JRE generates a runtime error. For example, consider the following interactive session:

```
Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> int x = 3;
> assert x == 3;
> assert x == 2;
```

```
AssertionError:
```

In DrJava's Interactions pane the user can continue; in an executing program the error terminates the program. Notice how the statement

```
assert x == 3;
```

has no effect. Quiet assertions are silent witnesses to the programmer's correct beliefs about the way the program is working. On the other hand, the assertion

```
assert x == 2;
```

fails, revealing the truth that even though we may believe `x` should have the value of 2 here, in fact it does not.

15.2 Unit Testing

Checks based on the `assert` statement (§ 15.1) are primitive devices for verifying the correctness of programs. *Unit testing* provides a more comprehensive framework for testing. Unit testing involves testing a component (that is, a unit) of a program. The class is the fundamental building block of a Java program. Clients can use a correctly written class to create objects that function correctly. If an object fails to behave correctly in a given situation, the error resides within the code of that object's class. The idea behind unit testing is that when all the components work correctly individually they can be combined into a complete software system that overall behaves correctly. Unit testing can be performed by hand or programmatically. Test-driven development (TDD) dictates that unit tests be automated and be available before the class to test is complete.

Hand testing is time consuming and error prone. Consider the following interactive session that tests a `TrafficLightModel` (Figure 10.1) object:

```

Interactions
Welcome to DrJava. Working directory is /Users/rick/java
> TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
> assert t.getState() == TrafficLightModel.CAUTION

```

A hand tester must remember to do this test after any change to the `TrafficLightModel` class and be sure that the state used on the first line of the interaction matches the state on the succeeding line. Couple this with the fact that perhaps hundreds of tests should be performed when any changes are made, and you can see why such hand testing is impractical in general.

A better approach would be to use a custom Java class to perform the testing as in `TrafficLightCustomTester` (Figure 15.1):

```

public class TrafficLightCustomTester {
    public void testColorChange() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assert TrafficLightModel.GO == t.getState();
        t.setState(TrafficLightModel.CAUTION);
        assert TrafficLightModel.CAUTION == t.getState();
        t.setState(182); // Bogus value
    }
}

```

```

        assert TrafficLightModel.STOP == t.getState();
    }
    public void testShow() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assert lt.show().equals("[ ( ) ( ) (G) ]");
        lt.setState(TrafficLightModel.CAUTION);
        assert lt.show().equals("[ ( ) (Y) ( ) ]");
        lt.setState(TrafficLightModel.STOP);
        assert lt.show().equals("[ (R) ( ) ( ) ]");
    }
    public static void main(String[] args) {
        TrafficLightCustomTester tester = new TrafficLightCustomTester();
        tester.testColorChange();
        tester.testShow();
    }
}

```

Listing 15.1: TrafficLightCustomTester—custom class for testing traffic light objects

If this program runs silently, all tests are passed. A runtime error indicates a failed test, as one of the assertions failed. The problem with the standalone custom testing class approach is there is no automated error reporting, and each tester is responsible for devising his own reporting scheme. Fortunately standardized testing frameworks are available that address these concerns.

15.3 JUnit Testing

JUnit is a testing framework developed by Erich Gamma and Kent Beck [1]. It is used to create unit tests for Java. DrJava provides built in support for JUnit. To see how it works, we will create a JUnit test case that subjects our traffic light classes to the same series of tests as performed in `TrafficLightCustomTester` (Figure 15.1). First, from the **File** menu choose the **New JUnit Test Case ...** item. Next, enter the name `TrafficLightTester` in the test case name dialog. The following skeleton code should appear in the editor:

```

import junit.framework.TestCase;

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when
 * running
 * the test with JUnit.
 */
public class TrafficLightTester extends TestCase {

    /**
     * A test method.
     * (Replace "X" with a name describing the test. You may write as

```

```

    * many "testSomething" methods in this class as you wish, and
    * each
    * one will be called when running JUnit over this class.)
    */
    public void testX() {
    }

}

```

The comments explain how you should edit the file to produce the desired test cases. We could change the name of `testX()` to `testColorChange()` and add code to test the state changing operation. The test methods can contain any Java code, but their purpose should be conducting tests on application objects.

Our `TrafficLightTester` class subclasses `junit.framework.TestCase` from which it inherits a great deal of functionality. The big benefit is the ability for the JUnit framework to create an instance of our `TrafficLightTester` class, run all of its test methods, and report the results in a standardized attractive way. All of this work performed by the framework is done automatically.

In `TrafficLightTester` (Figure 15.2), we write two test methods—`testColorChange()` and `testShow()`:

```

import junit.framework.TestCase;

public class TrafficLightTester extends TestCase {
    public void testColorChange() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assertEquals(TrafficLightModel.GO, t.getState());
        t.setState(TrafficLightModel.CAUTION);
        assertEquals(TrafficLightModel.CAUTION, t.getState());
        t.setState(182); // Bogus value
        assertEquals(TrafficLightModel.STOP, t.getState()); // Should be red
    }
    public void testShow() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ ( ) ( ) (G) ]");
        t.setState(TrafficLightModel.CAUTION);
        assertEquals(lt.show(), "[ ( ) (Y) ( ) ]");
        t.setState(TrafficLightModel.STOP);
        assertEquals(lt.show(), "[ (R) ( ) ( ) ]");
    }
}

```

Listing 15.2: `TrafficLightTester`—testing traffic light objects

After compiling the above code, select the **Test** option from the toolbar. You may also select the Test All Documents item from the Tools menu or press **Ctrl T** (**Cmd T** on a Mac). A program that is part of the JUnit framework is executed that creates an instance of your test class. This program executes each method that begins with the prefix `test`. We may add other methods that do not begin with `test`, but these can only serve as helper methods for the

test methods; the JUnit framework will not directly call these non-test methods. The results of the above test are shown in Figure 15.1: As the tests are run a green bar labeled Test Progress indicates the tests' progress. The green

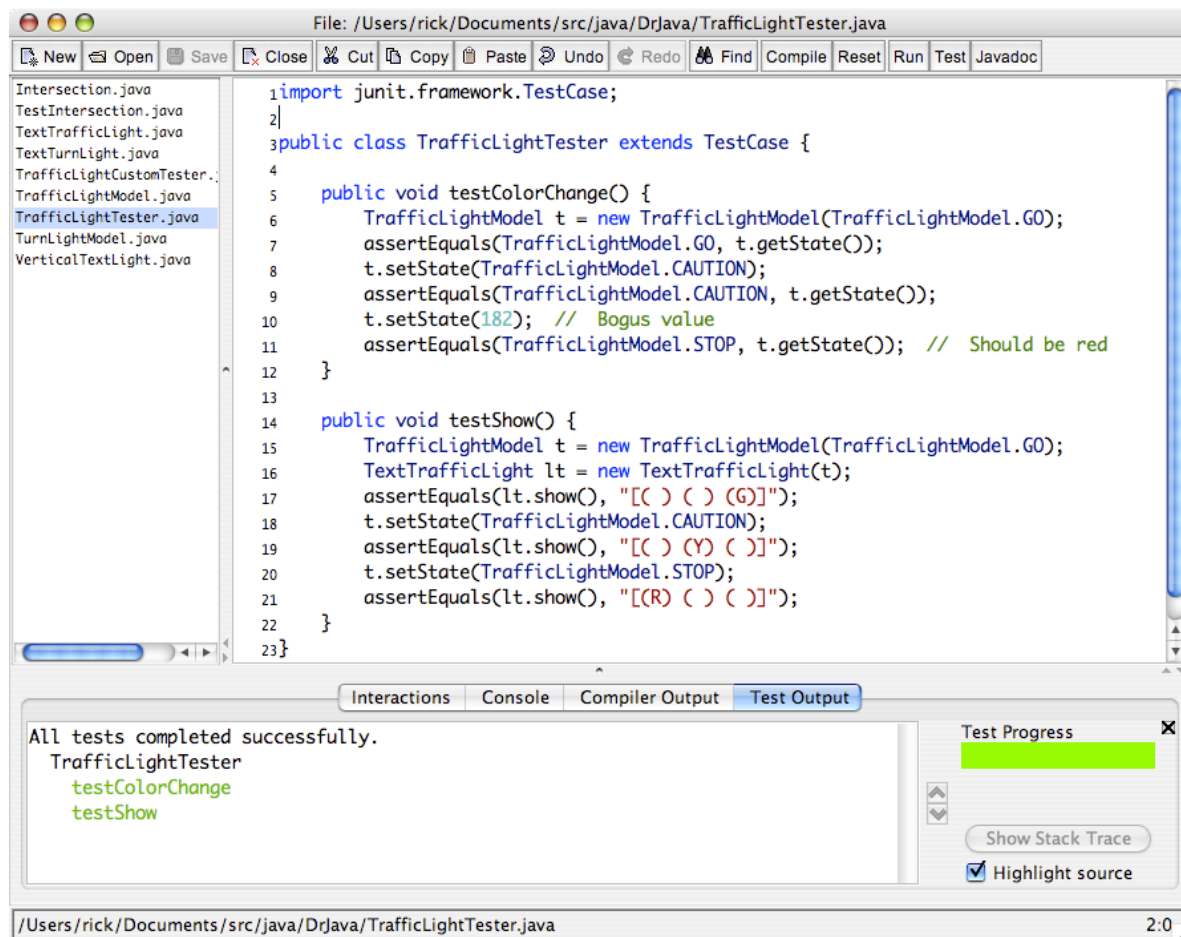


Figure 15.1: Results of a passed JUnit test

test method names shows the tests that passed. If you change Line 10 in the test to check for black instead of red, you will get the results shown in Figure 15.2: The bar turns red if a test fails. The offending test method is highlighted in red, while passing tests are green.

The `assertEquals()` method, not to be confused with the `assert` statement (see § 15.1), is the primary method that we will use in our tests. `assertEquals()` is inherited from `TestCase` and is overloaded to allow comparisons of all primitive and reference types. Its first parameter is the value expected by the tester; its second parameter is the actual value. The actual value is usually an expression to be evaluated (like `t.getState()`). Primitive types are compared as with `==`. Since all reference types have an `equals()` method (see § 14.2), the `assertEquals()` method compares reference types with `equals()`.

JUnit thus works like our original plain Java test program, but it provides an attractive report of the results of the tests. JUnit relieves developers from hand testing application classes.

While `TrafficLightTester` (Figure 15.2) is a valid test class, its design can be improved. It contains only two test methods, but it is checking six different things with `assertEquals()` calls. Each test method should be focused on checking one aspect of the object's functionality. The test method's name should be descriptive, reflecting the nature of the test being performed. Given a simple, focused test and a descriptive test name, when an `assertEquals()` fails, JUnit's report allows a tester to zoom in on the problem immediately. Also, more complex tests are difficult to maintain as the system evolves.

`BetterTrafficLightTester` (Figure 15.3) is a better test class, since it uses simpler, more focused test methods.

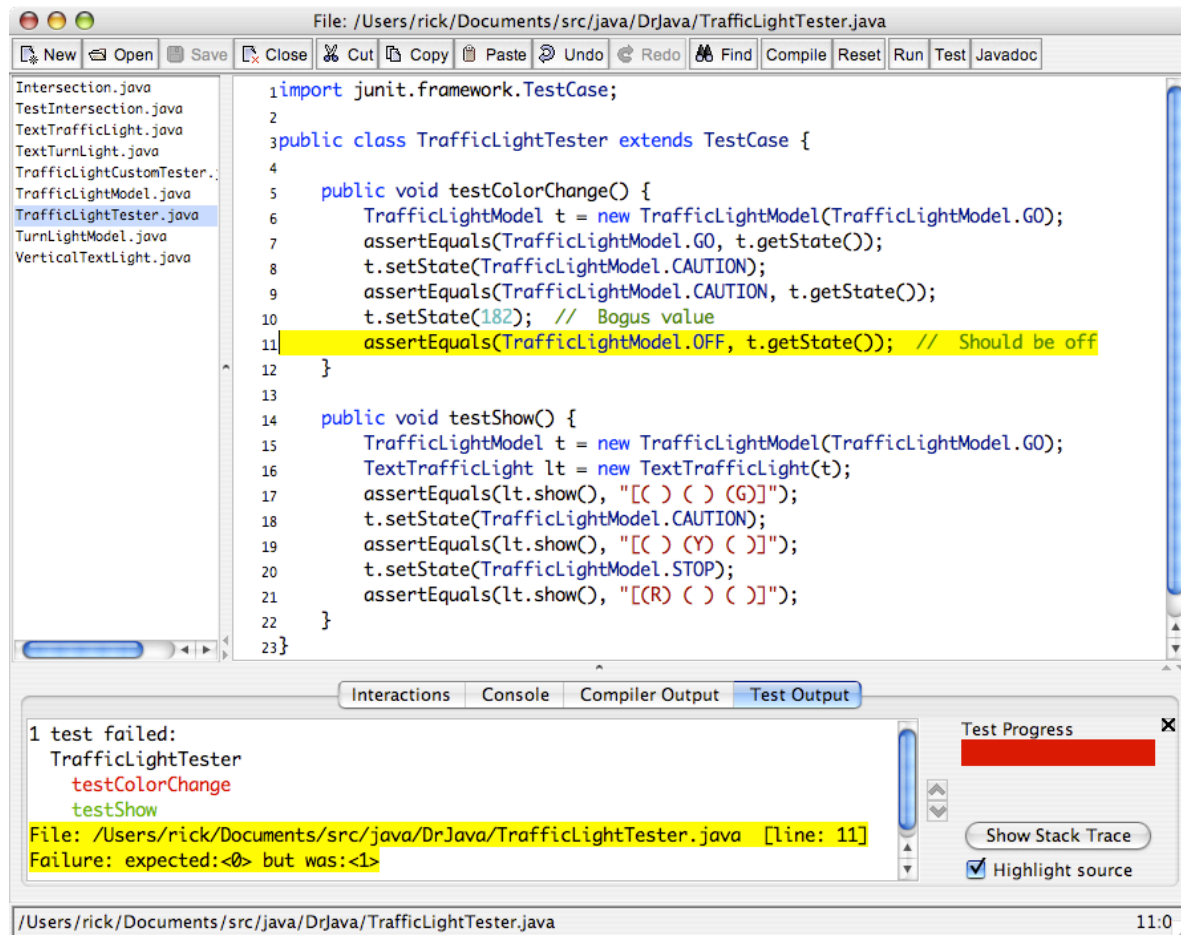


Figure 15.2: Results of a failed JUnit test

```
import junit.framework.TestCase;

public class BetterTrafficLightTester extends TestCase {
    public void testMakeGoLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        assertEquals(TrafficLightModel.GO, t.getState());
    }

    public void testMakeCautionLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        assertEquals(TrafficLightModel.CAUTION, t.getState());
    }

    public void testMakeStopLight() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        assertEquals(TrafficLightModel.STOP, t.getState());
    }

    public void testColorChangeGoToCaution() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        t.change();
        t.setState(TrafficLightModel.CAUTION);
    }
}
```

```

    }

    public void testColorChangeStopToGo() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        t.change();
        t.setState(TrafficLightModel.GO);
    }

    public void testColorChangeCautionToStop() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        t.change();
        t.setState(TrafficLightModel.STOP);
    }

    public void testStopString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.STOP);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ (R) ( ) ( ) ]");
    }

    public void testGoString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.GO);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ ( ) ( ) (G) ]");
    }

    public void testCautionString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.CAUTION);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ ( ) (Y) ( ) ]");
    }

    public void testOffString() {
        TrafficLightModel t = new TrafficLightModel(TrafficLightModel.OFF);
        TextTrafficLight lt = new TextTrafficLight(t);
        assertEquals(lt.show(), "[ ( ) ( ) ( ) ]");
    }
}

```

Listing 15.3: BetterTrafficLightTester—more focused unit tests

15.4 Regression Testing

Programs, especially large commercial systems, are built incrementally. This means that a program is written with parts missing and with incomplete parts. Such a system, if constructed properly, can still be compiled and run; for example, some methods may have empty bodies because their code has yet to be written. Such a work in

progress necessarily will have limited functionality. Testers can develop appropriate test cases that test this limited functionality. As parts are added or completed, the system's functionality increases. Testers then develop additional test cases to test these new features. I

In TDD, where tests are written before the functional code they are meant to test, code is written only as a result of a failed test. This philosophy is very different from our approach thus far. Want to add a new feature to an existing program? Write the test(s) for that feature and then test the existing program. Of course the test(s) will fail, because the feature has yet to be added! You must now write the code for that feature and work on that code until all the tests are passed. In TDD the tests are crucial, since they certify that a feature behaves acceptably and is considered "correct" with respect to all the tests. Tests can contain errors, since the act of writing the tests is itself a form of programming. Fortunately TDD recommends that each test be simple and focused, so simple in fact that the potential for errors is greatly reduced. Also, focused tests are more helpful in pinpointing the location of bugs. A test that tries to do too much is known as an *eager test*. An eager test might test more than one method of an object. Should that eager test fail, the guilty method would not be immediately known; on the other hand, a focused test, one testing only one method, upon failure would implicate directly the guilty method.

Unfortunately, since software systems can be so complex, it is not uncommon for the addition of a new feature to break existing features that worked before the addition of the new feature. The new feature is likely at fault, but if the new feature has been implemented correctly it may mean the pre-existing features have some lurking errors. If indeed some existing features have errors, it means the existing test cases for those features are incorrect or incomplete. New test cases must be added for those existing features to demonstrate their problems, and then the existing features can be corrected.

Since test-driven development mandates that tests are written early, features are added to a system only when tests are available to assess the correctness of those features. The tests for existing features are retained to ensure that the new features do not break existing features. Such a strategy is called *regression testing*. Test cases that apply to a given feature are removed from the test suite only if that feature is removed from the system. The number of test cases naturally continue to grow as functionality increases. As you can see, an automated testing framework such as JUnit is essential for managing regression testing.

15.5 Summary

- The `assert` statement checks at runtime the value of a Boolean expression.
- An assertion that evaluates to true does not affect a program's runtime behavior.
- An failed assertion results in a runtime error.
- Unit tests check the correctness of components in isolation from the rest of the system.
- The JUnit testing framework provides testers an automated way of performing unit tests, and it reports the results of the tests in a standard way.
- DrJava can generate a skeleton JUnit test case.
- JUnit's test methods begin with the prefix `test...`
- The `assertEquals()` methods of JUnit's `TestCase` class compares expected values to expressions. Successes and failures are tracked by the JUnit framework.

15.6 Exercises

1. What is the purpose of Java's `assert` statement?
2. How do you use Java's `assert` statement?
3. What are two courses of action that program execution can take when an `assert` statement is encountered?
4. What does TDD stand for?
5. What is one tenet of TDD that assists in writing correct tests?
6. What is unit testing?
7. Provide the JUnit `TestCase` method call that would check if integer variable `x` has the value 100.
8. What prefix must begin all JUnit test methods?
9. Does the `assertEquals()` method of the `TestCase` class have anything to do with Java's `assert` statement?

Chapter 16

Using Inheritance and Polymorphism

In this chapter we make use of inheritance and polymorphism to build a useful data structure.

16.1 Abstract Classes

Circle1a (Figure 16.1) is a variation of Circle1 (Figure 7.1) with the `circumference()` method renamed `perimeter()`.

```
public class Circle1a {
    private final double PI = 3.14159;
    private double radius;
    public Circle1a(double r) {
        radius = r;
    }
    public double perimeter() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 16.1: Circle1a—a slightly modified Circle1

Consider a new class—Rectangle1a (Figure 16.2). (The “1a” in its name simply implies that it is somehow associated with the Circle1a class.)

```
public class Rectangle1a {
    private double width;
    private double height;
    public Rectangle1a(double w, double h) {
```

```

        width = w;
        height = h;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2*width + 2*height;
    }
}

```

Listing 16.2: Rectangle1a—a stateful rectangle

Objects of these classes represent circles and rectangles—simple two-dimensional geometric shapes. We say they are *kinds of* shapes; that is, a circle is a *kind of* shape, and a rectangle is another *kind of* shape. As such, these specific kinds of shapes exhibit some common characteristics; for example, they both have areas and perimeters. We can *classify* these specific circle and rectangle shapes under the general category of *shapes*.

Suppose we want to write a method that would accept any kind of shape object and use the methods common to all shape objects to compute some useful result; for example, the perimeter-to-area ratio of an object, as in `PerimeterToAreaRatio` (Listing 16.3).

```

public class PerimeterToAreaRatio {
    public static double perimeterToAreaRatio(Shape s) {
        return s.perimeter()/s.area();
    }
}

```

Listing 16.3: `PerimeterToAreaRatio`—computes the perimeter-to-area ratio of a geometric shape

Observe that if we replace `Shape` in the parameter list of `perimeterToAreaRatio()` with `Circle1a`, it will work for `Circle1a` objects. Similarly, if we replace `Shape` with `Rectangle1a`, it will work for `Rectangle1a` objects. We do not want to write two different `perimeterToAreaRatio()` methods, one for each shape type. Instead, we really want only one method that works on both shape types, as shown here in the `PerimeterToAreaRatio` class. Inheritance provides the way to make this happen.

In OOP, we use *class hierarchies* formed by inheritance relationships to classify kinds of objects. A Java class represents a kind of an object. When two classes share properties and are related to each other conceptually, we often can create a new class exhibiting those common properties. We then modify the two original classes to make them subclasses of the new class. In our example, circles have areas and perimeters, and rectangles have areas and perimeters. Both are kinds of shapes, so we can make a new class named `Shape` and make `Circle1a` and `Rectangle1a` subclasses of `Shape`.

How do we capture the notion of a general, nonspecific shape in a Java class? Here are our requirements:

- We would name the class `Shape`.

- Any object that is a kind of `Shape` should be able to compute its area and perimeter through methods named `area()` and `perimeter()`. (`Circle` and `Rectangle` objects both qualify.)
- We cannot write actual code for the `area()` and `perimeter()` methods in our generic `Shape` class because we do not have a specific shape. No universal formula exists to compute the area of any shape because shapes are so different.
- Clients should be unable to create instances of our `Shape` class. We can draw a circle with a particular radius or a rectangle with a given width and height, but how do we draw a generic shape? The notion of a *pure* shape is too abstract to draw. Likewise, our `Shape` class captures an abstract concept that cannot be materialized into a real object.

Java provides a way to define such an abstract class, as shown in `Shape` (Figure 16.4).

```
public abstract class Shape {  
    public abstract double area();  
    public abstract double perimeter();  
}
```

Listing 16.4: `Shape`—a generic shape

In `Shape` (Figure 16.4):

- The class is declared `abstract`. Such a class is called an *abstract class*. We are unable to create instances of an abstract class, just as we really cannot have a plain shape object that is not some kind of specific shape.
- An abstract class may have zero or more methods that are declared `abstract`. Abstract methods have no bodies; a semicolon appears where the body of a non-abstract method would normally go. We are *declaring* the method but not *defining* it. Since we have no concrete information about a generic shape we cannot actually compute area and perimeter, so both of these methods are declared `abstract`.
- While not shown here, an abstract class may contain concrete (that is, non-abstract) methods as well. An abstract class can have non-abstract methods, but a non-abstract class may not contain any abstract methods.

A non-abstract class is called a *concrete class*. All classes are either abstract or concrete. All the classes we considered before this chapter have been concrete classes. A non-abstract method is called a concrete method. A concrete class may not contain abstract methods, but an abstract class may contain both abstract methods and concrete methods.

Given our `Shape` class we can now provide the concrete subclasses for the specific shapes in `Circle` (Figure 16.5) and `Rectangle` (Figure 16.6).

```
public class Circle extends Shape {  
    private final double PI = 3.14159;  
    private double radius;  
    public Circle(double r) {  
        radius = r;  
    }  
}
```

```

    public double perimeter() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}

```

Listing 16.5: Circle—a circle subclass

```

public class Rectangle extends Shape {
    private double width;
    private double height;
    public Rectangle(double w, double h) {
        width = w;
        height = h;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * width + 2 * height;
    }
}

```

Listing 16.6: Rectangle—a rectangle subclass

Now `PerimeterToAreaRatio` (Figure 16.3) works as is on any `Shape` subclass. Polymorphism enables the `perimeterToAreaRatio()` to execute the proper `area()` and `perimeter()` code for the object at hand. Figure 16.1 shows the UML diagram for our shapes class hierarchy. Abstract class names are shown in italics in the UML class diagrams.

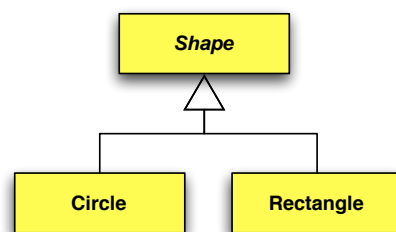


Figure 16.1: Class hierarchy of shape classes. It is common practice to omit the superclass of `Shape`; that is, the `java.lang.Object` class is not shown here.

Suppose we must develop an inventory program for a produce market. One of the market's specialties is fresh fruit, including bananas and several varieties of apples and grapes. For example, we might have a class for golden

delicious apples and another for granny smith apples. We note that both varieties have some common properties (after all, they are both kinds of apples). It makes sense to create a common superclass, `Apple`, and derive the `GoldenDelicious` and `GrannySmith` classes from `Apple`, specializing the subclasses as required. Classes for red and white grapes could have `Grape` as a common superclass. Further, all fruit have some common properties, like color (red, yellow, etc.), taste (sweet, tangy, etc.), and size (volume or mass). Given this scenario we can create a hierarchy of classes as illustrated by the UML diagram in Figure 16.2. Ordinarily, we use classes as a blueprint

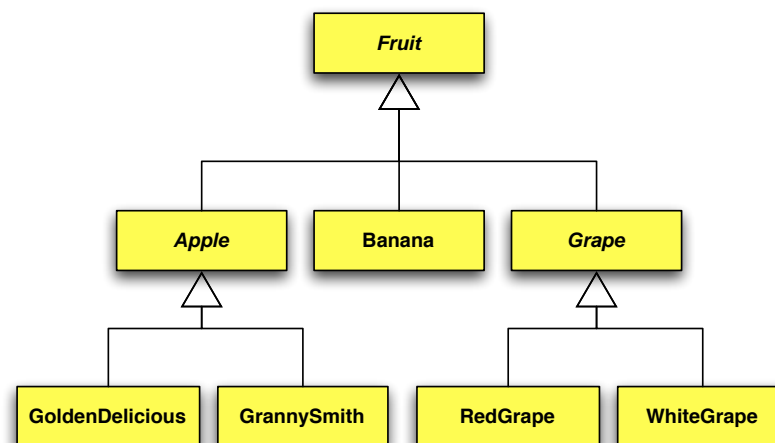


Figure 16.2: Class hierarchy of fruit classes.

for creating objects. We note that while golden delicious apples and granny smith apples would have a place in the inventory database, generic “apples” would not.

To recap, when two or more classes have common characteristics, and the classes conceptually represent specific examples of some more general type, we can use inheritance to factor out their common properties. We do so as follows:

- Make a new superclass that contains the common characteristics.
- Declare the class `abstract` if it represents an abstract concept and actual objects of this class do not make sense because there is not enough specific information to implement all of its required functionality.
- Modify the existing specific classes to be subclasses of the new superclass.

The new superclass is meant to distill the common characteristics of its subclasses. It represents the abstract notion of what it means to be instances of any of its subclasses.

16.2 A List Data Structure

Armed with our knowledge of inheritance, polymorphism, Java’s wrapper classes (§ 13.4), and abstract classes, we can build a versatile list data structure. First, we need to think about the nature of lists:

- A list holds a collection of items in a *linear sequence*. Every nonempty linear sequence has the following properties:
 - there exists a unique first item in the list
 - there exists a unique last item in the list

- every item except the first item has a unique predecessor
- every item except the last item has a unique successor
- An empty list contains nothing. One empty list is indistinguishable from any other empty list. It follows that there should be exactly one empty list object, because making more than one does not provide any advantages.
- We can view any nonempty list as the first item followed by a list (the rest of the list). A nonempty list has at least one item. In a list containing only one item, the rest of the list refers to the empty list.
- It is easy to determine the number of items in a list:
 - If the list is empty, the number of items it contains is zero.
 - If the list is nonempty, the number of items it contains is one (for its first item) plus the number of items in the rest of the list.

As we shall see, this recursive description is readily translated into a recursive method.

- It is easy to add an item to the end of a list:
 - If the list is empty, we just make a new nonempty list whose first item is the item we wish to add. The rest of this new list is the empty list.
 - If the list is nonempty, we simply add the new item to the rest of this list.

This recursive algorithm also is easily translated into Java code.

As you can see, we have two distinct kinds of lists: empty lists and nonempty lists. We have to be able to treat both empty and nonempty lists as just *lists* even though empty lists and nonempty lists are functionally different kinds of objects. We need to distill the properties common to all kinds of lists into an abstract list class. From this abstract list class we will derive the concrete subclasses for empty and nonempty list objects. Figure 16.3 shows the resulting class hierarchy.

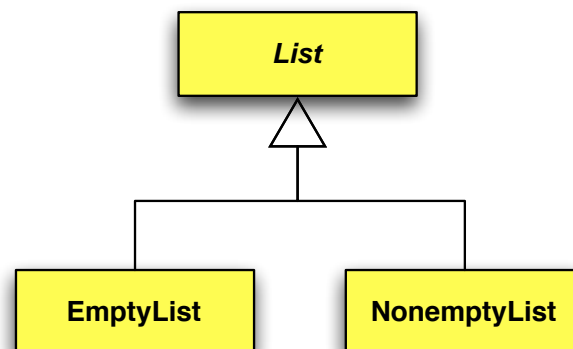


Figure 16.3: Class hierarchy of list classes.

List (Figure 16.7) provides good starting point:

```

public abstract class List {
    public abstract int length();
    public abstract List append(Object newElement);
    public String toString() {

```

```

        return "[" + toStringHelper() + "]";
    }
    protected abstract String toStringHelper();
}

```

Listing 16.7: List—Abstract superclass for list objects

As we can see in List (Figure 16.7):

- Any list object can determine its length; that is, the number of elements it holds.

```
public abstract int length();
```

We cannot specify how this method should work since empty and nonempty lists will do this differently; therefore, this method is abstract.

- New items can be appended to any list object.

```
public abstract List append(Object newElement);
```

Again, we do not have enough detail in a generic list to indicate exactly how this is to be done, so `append()` is declared abstract. The parameter type of `append()` is `Object`. Since all classes are subclasses of `Object` (§ 14.1), either directly or indirectly, this means any reference type can be added to our list. Since all the primitive types have wrapper classes (§ 13.4), any primitive type as well can be added to our list. This means anything can be added to a list, even another list!

The return type is `List`, but when a subclass overrides the `append()` method it will actually return a reference to an instance of a subclass of `List`. Because of the *is a* relationship, this is legal. For all we know now, the `append()` method returns a reference to some kind of list object.

- Every list object can be rendered as a string in the same way: The first character is `[` and the last character is `]`. The characters in between are determined by another method, `toStringHelper()`. Since actual code can be written for the `toString()` method, this method is concrete, not abstract.
- The `toStringHelper()` method is abstract, since for a generic list object we cannot know how to access its contents, if indeed it has any contents. It uses a new access privilege—`protected`. A `protected` member is accessible to code in subclasses and classes within the same package, but inaccessible to all other classes. This is exactly the level of protection we need here. `toStringHelper()` is meant to be an internal helper method; it is not meant to be used by clients. Making it `private`, however, would make it inaccessible to subclasses, meaning that subclasses could not override it. Since it is declared abstract, concrete subclasses *must* override it. The `protected` specifier essentially means `public` to subclasses and classes within the same package and `private` to all other classes.

Since `List` is abstract, we cannot create actual `List` objects. `List` serves as the superclass for two concrete subclasses that we will use to build real lists. The first is `EmptyList` (Figure 16.8):


```

public class EmptyList extends List {
    // This constant is used when an empty list is needed
    public static final List EMPTY = new EmptyList();

    private EmptyList() {} // Cannot create an instance of EmptyList

    public int length() {
        return 0;
    }
    public List append(Object newElement) {
        return new NonemptyList(newElement, EMPTY);
    }
    protected String toStringHelper() {
        return " ";
    }
}

```

Listing 16.8: EmptyList—Class for empty list objects

Despite its small size, EmptyList (Listing 16.8) is full of interesting features:

- EmptyList is a subclass of List,

```
public class EmptyList extends List {
```

but it is concrete; therefore, it may not contain any abstract methods. As we determined above, however, it should be trivial to add the needed functionality to empty lists.

- A class constant consisting of an EmptyList instance

```
public static final List EMPTY = new EmptyList();
```

is made available to clients for general use. As indicated in the next item, any reference to an empty list *must* use this constant.

- The only constructor is private.

```
private EmptyList() {} // Cannot create an instance of EmptyList
```

As the comment says, this means clients are not allowed to create EmptyList objects directly using the new operator. For example, the statement

```
EmptyList e = new EmptyList(); // Compiler error!
```

appearing anywhere outside of this class will result in a compiler error. This fact coupled with the EMPTY EmptyList constant means that exactly one EmptyList object will ever exist when this class is used. We say that EMPTY is a *singleton* empty list object.

- Empty lists trivially have no elements:

```
public int length() {
    return 0;
}
```

so the length of an empty list is always zero.

- Appending a new item to an empty list results in a new nonempty list:

```
public List append(Object newElement) {
    return new NonemptyList(newElement, EMPTY);
}
```

We will see the `NonemptyList` class next. Its constructor takes two arguments:

- The first argument is the first item in the list.
- The second argument is the rest of this new nonempty list.

Observe that the net effect is to make a list containing only one element—exactly what we need when appending to an empty list.

- Converting the contents of the empty list to a string is also trivial.

```
protected String toStringHelper() {
    return " ";
}
```

The `toStringHelper()` method simply returns a space—there are no elements to show.

- The `toString()` method is inherited from `List`. It ensures the contents of the resulting string are bracketed within `[]`. This superclass `toString()` method calls `toStringHelper()` polymorphically, so the correct `toStringHelper()` is called for the exact type of the list object.

Even though `EmptyList` is a concrete class, we cannot use it until we implement `NonemptyList` (▮ 16.9):

```
public class NonemptyList extends List {
    private Object first;
    private List rest;

    public NonemptyList(Object f, List r) {
        first = f;
        rest = r;
    }

    public int length() {
        return 1 + rest.length();
    }
}
```

```

public List append(Object newElement) {
    return new NonemptyList(first, rest.append(newElement));
}

protected String toStringHelper() {
    return " " + first + rest.toStringHelper();
}
}

```

Listing 16.9: NonemptyList—Class for nonempty list objects

In NonemptyList (Listing 16.9):

- Two private fields:

```

private Object first;
private List rest;

```

capture our original definition of what constitutes a nonempty list:

- Every nonempty list has a first item (*first*), and
- every nonempty list has the rest of the list (*rest*) that follows the first item.
- The type of *first* is `Object`. The practical result is any type of data can be assigned to the *first* instance variable.
- The type of *rest* is `List`. This means references to both `EmptyList` objects and `NonemptyList` objects can be assigned to the *rest* instance variable. A list that contains only one item would assign the item to *first* and then assign *rest* to the empty list.
- The constructor initializes the instance variables:

```

public NonemptyList(Object f, List r) {
    . . .
}

```

- The `length()` method

```

public int length() {
    return 1 + rest.length();
}

```

works exactly as we originally specified. The length of a nonempty list is one (for its first item) plus the length of the rest of the list. Note the recursive nature of this method. It is up to the remainder of the list (referenced by *rest*) to compute its own length. Empty lists will report zero, and nonempty lists call this same method polymorphically but with a different object.

- The `append()` method

```

public List append(Object newElement) {
    return new NonemptyList(first, rest.append(newElement));
}

```

simply creates a new list object. We know for sure the type of list to create is a `NonemptyList` since we are appending an item onto the current (nonempty) list. The first item in the new list will be the same as the first item in this list. The rest of the new list is the result of appending the new item to the end of the current rest of the list. The recursive nature of the method call leads eventually to an attempt to append the new item to the empty list. The recursion stops there (the `append()` method in `EmptyList` is definitely nonrecursive), and a new nonempty list containing the new item is returned.

- The `toStringHelper()` method

```

protected String toStringHelper() {
    return " " + first + rest.toStringHelper();
}

```

is another recursive procedure. We append the first item and then defer the rest of the task to the `rest` object.

- As with `EmptyList`, the `toString()` method is inherited from `List`. When this superclass `toString()` method is called on behalf of a `NonemptyList` object, `toString()` calls `NonemptyList`'s `toStringHelper()` polymorphically.

We now have everything in place to experiment with our new list data structure:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> e = new EmptyList(); // Attempt to create an empty list
IllegalAccessException: Class
koala.dynamicjava.interpreter.context.GlobalContext can not access a
member of class EmptyList with modifiers "private"
    at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:486)
> e = EmptyList.EMPTY; // Do it the intended way
> e
[ ]
> list1 = e.append("Fred");
> list1
[ Fred ]
> list2 = list1.append("Wilma");
> list2
[ Fred Wilma ]
> list3 = list2.append(19.37);
> list3
[ Fred Wilma 19.37 ]
> list4 = list3.append(15);
> list4
[ Fred Wilma 19.37 15 ]
> list5 = list4.append("Barney");
> list5

```

```
[ Fred Wilma 19.37 15 Barney ]
> list5.length()
5
```

Figure 16.4 illustrates the data structure referenced by `list5` as created by this sequence of interactions. If the

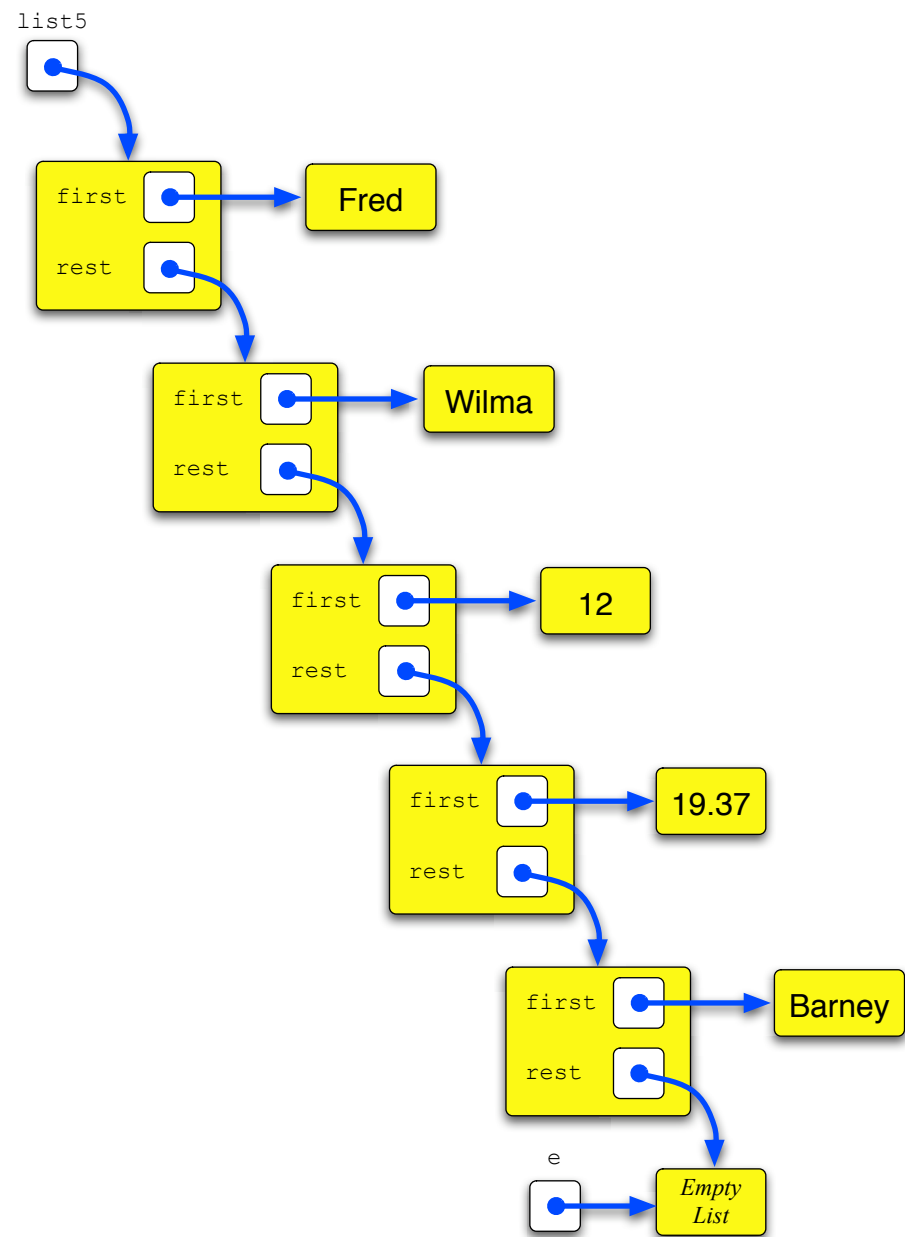
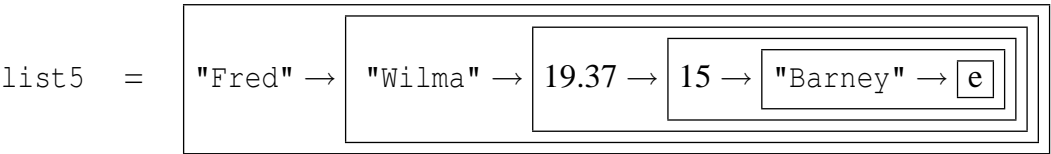
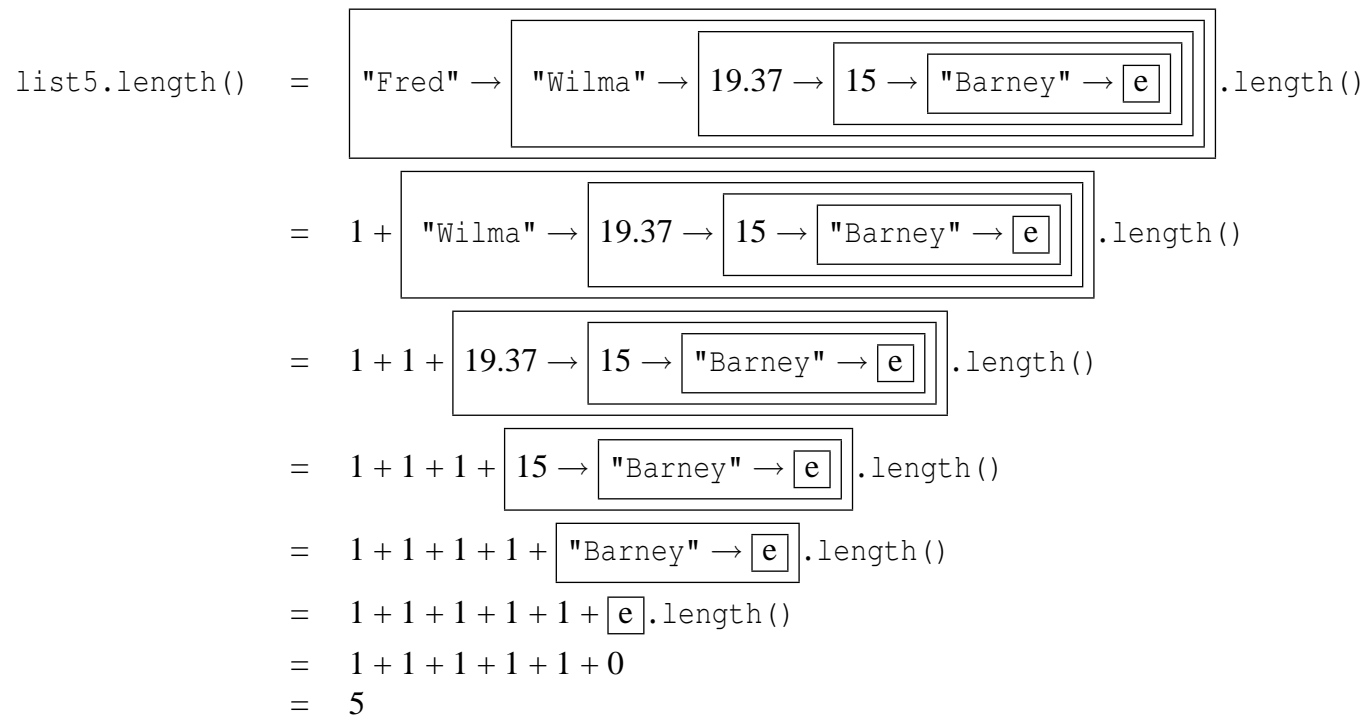


Figure 16.4: Data structure created by the interactive sequence

structure of list `list5` is represented by the following:



then the activity performed by the call `list5.length()` is



The compiler automatically creates the appropriate wrapper objects for the primitive types 19.37 and 15.

Like Java's `String` objects, our list objects are immutable. Clients cannot modify the contents of a list. Our `append()` method does not modify an existing list; it creates a new list with the new item added on the end. The original list is unaffected by the action. What if we wish to modify a list? We can simply reassign the list reference, as is:

```
list5 = list5.append(2);
```

In this case the original list of `list5` is replaced by the new list created by the `append()` method call.

As an interesting side note, since our lists can hold any kinds of objects, we can easily have lists of lists:

Interactions

```

Welcome to DrJava. Working directory is /Users/rick/java
> list = EmptyList.EMPTY;
> list
[ ]
> list = list.append(19.37);
> list
[ 19.37 ]
> list = list.append("Fred");
> list = list.append("Wilma");
> list
[ 19.37 Fred Wilma ]
> list.length()
3
> list = list.append(list);
> list
[ 19.37 Fred Wilma [ 19.37 Fred Wilma ] ]
> list.length()
4

```

The length of the whole list at the end is four since the last element (itself a list) is a single object (a `NonemptyList` object).

`ListTest` (Figure 16.10) provides some lightweight testing for our list data structure.

```
import junit.framework.TestCase;

public class ListTest extends TestCase {

    public void testAppendMethod() {
        List list = EmptyList.EMPTY;
        assertEquals("[ ]", list.toString());
        assertEquals(0, list.length());
        list = list.append(22);
        //System.out.println(list);
        assertEquals("[ 22 ]", list.toString());
        assertEquals(1, list.length());
        list = list.append("Fred");
        assertEquals("[ 22 Fred ]", list.toString());
        assertEquals(2, list.length());
        list = list.append(2.2);
        assertEquals("[ 22 Fred 2.2 ]", list.toString());
        assertEquals(3, list.length());
        list = list.append("Wilma");
        assertEquals("[ 22 Fred 2.2 Wilma ]", list.toString());
        assertEquals(4, list.length());
    }
}
```

Listing 16.10: `ListTest`—JUnit test file

To conclude, it is important to note that the immutable list implementation presented here is simple and elegant but not very efficient. A nonrecursive, mutable version presented in § 18.4 is more efficient. We will consider this more efficient implementation after we cover iteration in Chapter 17.

16.3 Interfaces

Java provides a construct that is similar to an abstract class with methods that are exclusively `public` and `abstract`. An interface specifies a collection of methods that any class that is to comply with that interface must implement. Since `Shape` (Figure 16.4) is an abstract class containing no instance variables and no concrete methods, it can be expressed as an interface, as shown in `IShape` (Figure 16.11).

```
public interface IShape {
    double area();
    double perimeter();
}
```

Listing 16.11: IShape—an interface for shape objects

From the IShape interface we can now specify that particular classes comply with the interface using the `implements` keyword, as shown in Ring (Listing 16.12) and Quadrilateral (Listing 16.13).

```
public class Ring implements IShape {
    private final double PI = 3.14159;
    private double radius;
    public Ring(double r) {
        radius = r;
    }
    public double perimeter() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

Listing 16.12: Ring—a ring (circle) class implementing the shape interface

```
public class Quadrilateral implements IShape {
    private double width;
    private double height;
    public Quadrilateral(double w, double h) {
        width = w;
        height = h;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * width + 2 * height;
    }
}
```

Listing 16.13: Quadrilateral—a quadrilateral (rectangle) class implementing the shape interface

The Ring and Quadrilateral classes both implement the IShape interface. They must contain at least the methods defined in IShape, but implementing classes can add additional methods and add fields.

Methods defined in interfaces are implicitly abstract and public; thus, `area()` and `perimeter()` within IShape are public abstract methods. An interface is like an abstract class in that it is not possible to create instances of interfaces. Such a statement is illegal:


```
IShape shape = new IShape(); // Illegal to create an instance
```

an interface cannot contain instance variables or general class variables but can define constants (final static fields).

An interface serves as a contract that classes must fulfill that claim to comply with that interface. A class declares that it implements an interface with the `implements` keyword followed by a list of interfaces with which it complies. Unlike subclassing, where a single class cannot have more than one superclass, a class may implement multiple interfaces. In fact, any programmer-defined class *will* definitely extend a class (`Object`, if nothing else) and also *may* implement one or more interfaces.

Like a class, an interface defines a type; an implementing class is a subtype of the interface, just like a subclass is a subtype of its superclass. A special relationship exists between an interface and a class that implements that interface. This relationship is a *can do* relationship because an interface expresses what *can be done* in terms of method signatures. The interface specifies *what* can be done but cannot express *how* it is to be done. The how can only be specified in a method body, but in an interface all methods must be abstract. In practice, however, the *can do* relationship behaves like the *is a* relationship; that is, an object that is a subtype of an interface can be used in any context that expects the interface type. `ShapeReport` (Figure 16.14) shows how this subtyping relationship works.

```
public class ShapeReport {
    public static void report(IShape shape) {
        System.out.println("Area = " + shape.area()
                           + ", perimeter = " + shape.perimeter());
    }
    public static void main(String[] args) {
        IShape rect = new Quadrilateral(10, 2.5),
        circ = new Ring(5.0);
        report(rect);
        report(circ);
    }
}
```

Listing 16.14: `ShapeReport`—Illustrates the *is a* relationship for interfaces

Notice two things in `ShapeReport`:

- The `report()` method in `ShapeReport` expects an `IShape` type and works equally well with `Quadrilateral` and `Ring` objects. This is because of the *can do* relationship between an interface and its implementing class.
- In `main()`, the declared type of both `rect` and `circ` is `IShape`. While a literal `IShape` object cannot be created, a `Quadrilateral` object can be assigned to an `IShape` reference. Likewise, a `Ring` object can be assigned to an `IShape` reference. Again, this is because of the *can do* relationship between the implementing classes and `IShape`.

The reason that the parameter passing and assignment works is that, just as a subclass reference can be assigned to a superclass reference, a reference to a subtype can be assigned to a supertype reference. In fact the subclass-superclass relationship is a special case of the subtype-supertype relationship: the superclass of a class is its supertype, and a subclass of a class is a subtype of that class.

16.4 Summary

- A Java class represents a kind of an object, and a common superclass is used to organize different kinds of objects with common characteristics into the same category.
- An abstract method has no body.
- Concrete subclasses must override inherited abstract methods.
- A concrete class may not contain any abstract methods.
- An abstract class contains zero or more abstract methods.
- An abstract class may contain concrete methods.
- In a UML diagram, the name of an abstract class is italicized, and the name of a concrete class is not italicized.

16.5 Exercises

1. What reserved word specifies a class to be abstract?
2. Can instances be made of an abstract class?
3. Can all the methods in an abstract class be concrete?
4. Can any methods in a concrete class be abstract?
5. In Figure 16.2, which classes are abstract and which are concrete?
6. Explain how the `append()` method in the `NonemptyList` class correctly adds new elements to the end of the list.
7. Add the following methods to the `List` class and each of its subclasses as necessary:
 - (a) `public List prepend(Object newElement)`— inserts a new element onto the front of the list
 - (b) `public List concat(List other)`— splits two lists together in a manner analogous to string concatenation.
 - (c) `public boolean contains(Object seek)`— returns `true` if `seek` is found in the list; otherwise, returns `false`. The `equals()` should be used to check for equality.
 - (d) `public boolean equals(List other)`— determines if two lists contain the same elements (using their `equals()` methods) in the same order.
 - (e) `public Object get(int index)`— returns the item in position `index`. The first element in a nonempty list is at index zero. The method should return `null` if an invalid index is given:
 - Any index is invalid for an empty list.
 - In a nonempty list with n elements, a valid index is in the range $0 \leq \text{index} < n$.
8. Use list object to record the states that a traffic light assumes. This will keep a log of all the states that a traffic light enters. Make a new traffic light type named `LoggedTrafficLight`, a subclass of `TextTrafficLight` (Figure 10.2). Every call to its `setState()` method should append the requested state to the end of the list. Add a method named `reviewLog()` that simply prints out, in order, the states assumed by the traffic light since its creation. You should use but **not** modify the code of `TrafficLightModel` (Figure 10.1) and `TextTrafficLight` (Figure 10.2).

Chapter 17

Iteration

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems. Iteration and conditional execution form the basis for algorithm construction.

17.1 The `while` Statement

CountToFive (Figure 17.1) counts to five by printing a number on each output line.

```
public class CountToFive {  
    public void run() {  
        System.out.println(1);  
        System.out.println(2);  
        System.out.println(3);  
        System.out.println(4);  
        System.out.println(5);  
    }  
}
```

Listing 17.1: CountToFive—Simple count to five

In an Interactions pane we see the results:

```
Interactions  
Welcome to DrJava. Working directory is /Users/rick/java  
> new CountToFive().run();  
1  
2  
3  
4  
5  
>
```

How would you write the code for `CountToTenThousand`? Would you copy, paste, and modify 10,000 `System.out.println()` statements? That would be impractical! Counting is such a common activity, and computers routinely count up to very large values, so there must be a better way. What we really would like to do is print the value of a variable (call it `count`), then increment the variable (`count++`), and repeat this process until the variable is large enough (`count == 5`). This concept is known as *iteration*, and Java has a simple statement that supports it.

`IterativeCountToFive` (Figure 17.2) uses a `while` statement to count to five:

```
public class IterativeCountToFive {
    public void run() {
        int count = 1;                // Initialize counter
        while (count <= 5) {
            System.out.println(count); // Display counter, then
            count++;                   // Increment counter
        }
    }
}
```

Listing 17.2: `IterativeCountToFive`—Better count to five

`IterativeCountToFive` uses a `while` loop inside its `run()` method to display a variable that is counting up to five. Unlike with the approach taken in `CountToFive`, it is trivial to modify `IterativeCountToFive` to count up to 10,000 (just change the literal value 5 to 10000).

The statements

```
System.out.println(count); // Display counter, then
count++;                  // Increment counter
```

in `IterativeCountToFive` constitute the body of the `while` statement. The statements in the body are repeated over and over until the Boolean condition

```
count <= 5
```

becomes false.

The `while` statement has the general form:

```
while ( condition )
    body
```

- The reserved word `while` identifies an `while` statement.
- *condition* is a Boolean expression that determines whether or not the body will be (or will continue to be) executed. The condition must be enclosed within parentheses.
- *body* is like the `if` statement body (Section 5.2)—it can consist of a single statement or a block of statements.

Except for the reserved word `while` instead of `if`, `while` statements look identical to `if` statements. Often beginning programmer confuse the two or accidentally type `if` when they mean `while` or vice-versa. Usually the very different behavior of the two statements reveals the problem immediately; however, sometimes, in deeply nested complex logic, this mistake can be hard to detect.

Figure 17.1 shows how program execution flows through a `while` statement.

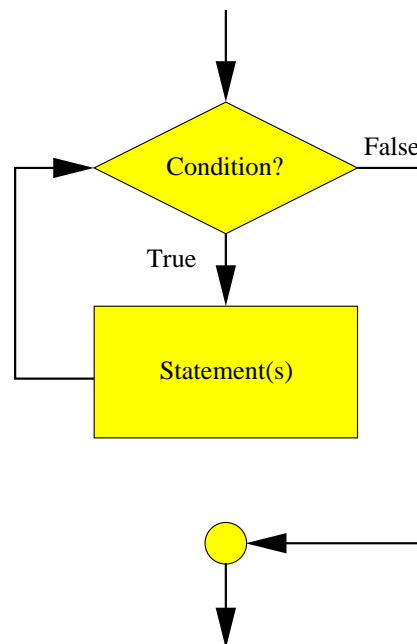


Figure 17.1: Execution flow in a `while` statement

The condition is checked before the body is executed, and then each time after the body has been executed. If the condition is false or becomes false, the body is not executed and the loop terminates. Observe that the body may never be executed if the Boolean expression in the condition is initially false.

`AddUpNonnegatives` (Figure 17.3) is a program that allows a user to enter any number of nonnegative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the nonnegative values. If a negative number is the first entry, the sum is zero.

```

import java.util.Scanner;

public class AddUpNonnegatives {
    public static void main(String[] args) {
        int input = 0,
            sum = 0;
        Scanner scan = new Scanner(System.in);
        while (input >= 0) {
            input = scan.nextInt();
            if (input >= 0) {
                sum += input;
            }
        }
        System.out.println("Sum = " + sum);
    }
}
  
```

Listing 17.3: AddUpNonnegatives—Sums any number of nonnegative integers

The initialization of `input` to zero guarantees that the body of the `while` loop will be executed at least once. The `if` statement ensures that a negative entry will not be added to `sum`. (Could the condition have used `>` instead of `>=` and achieved the same results?) Upon entry of a negative value, `sum` will not be updated and the condition will no longer be true. The loop terminates and the `print` statement is finally executed.

`AddUpNonnegatives` (Figure 17.3) shows that a `while` loop can be used for more than simple counting. `AddUpNonnegatives` does not keep track of the number (count) of values entered. The program simply accumulates the entered values in the variable named `sum`.

It is a little awkward that the same condition appears twice, once in the `while` and again in the `if`. The code `AddUpNonnegatives` (Figure 17.3) can be simplified somewhat with some added insight into the assignment statement. A simple assignment statement such as

```
x = y;
```

is actually an expression that has a value. Its value is the same as the value assigned to the variable. That means the following statement is possible:

```
x = (y = z);
```

Here, the value of `z` is assigned to `y` via the assignment `y = z`. This assignment expression itself has the value of `z`, so `z`'s value is assigned also to `x`. Unlike the arithmetic binary operators (`+`, `-`, `*`, etc.) which apply from left to right, the assignment operator associates from right to left. This means the parentheses can be omitted:

```
x = y = z;
```

The effect of this statement is that all the variables end up with the same value, `z`'s value. This extended assignment statement is sometimes called a *chained assignment*.

This curious fact about assignment can be put to good use to shorten our `AddUpNonnegatives` code:

```
import java.util.Scanner;

public class AddUpNonnegativesSimpler {
    public static void main(String[] args) {
        int input, sum = 0;
        Scanner scan = new Scanner(System.in);
        while ((input = scan.nextInt()) >= 0) {
            sum += input;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Listing 17.4: AddUpNonnegativesSimpler—Sums any number of nonnegative integers

In `AddUpNonnegativesSimpler` (Figure 17.4):

- The variable `input` no longer must be initialized to zero, because it is assigned before its value is checked by the `while`. The value of `sum` must still be initialized since it is accumulating the sum and must begin with a zero value.
- In the expression

```
(input = scan.nextInt()) >= 0
```

the assignment is performed first, then the value of the assignment is compared to zero.

- The modification of `sum` can only be performed as long as the condition remains true. The update statement never can be executed whenever `input` is negative; thus, the separate `if` statement is no longer needed.

Some programmers consider the simplified version tricky and prefer the first version. The second version is slightly more efficient since the condition is checked only one time through the loop instead of two. In this case, however, the speed difference is negligible, so the less tricky design is acceptable.

17.2 Nested Loops

Just like in `if` statements, `while` bodies can contain arbitrary Java statements, including other `while` statements. A loop can therefore be nested within another loop. `TimesTable` (Figure 17.5) prints a multiplication table on the screen using nested `while` loops.

```
public class TimesTable {
    public static void main(String[] args) {
        // Print a multiplication table to 10 x 10
        int row = 1;
        // Print column heading
        System.out.println("      1   2   3   4   5   6   7   8   9  10");
        System.out.println(" +-----+");
        while ( row <= 10 ) {                // Table has ten rows.
            System.out.printf("%3d|", row); // Print heading for this row.
            int column = 1;                  // Reset column for each row.
            while ( column <= 10 ) {          // Table has ten columns.
                System.out.printf("%4d", row*column);
                column++;
            }
            row++;
            System.out.println(); // Move cursor to next row
        }
    }
}
```

Listing 17.5: `TimesTable`—Prints a multiplication table

The output of `TimesTable` is

	1	2	3	4	5	6	7	8	9	10
+-----										
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

This is how `TimesTable` works:

- It is important to distinguish what is done only once (outside all loops) from that which is done repeatedly. The column heading across the top of the table is outside of all the loops; therefore, it is printed all at once.
- The work to print the heading for the rows is distributed throughout the execution of the outer loop. This is because the heading for a given row cannot be printed until all the results for the previous row have been printed.
- `System.out.printf()` is used to print the table contents so that the entries are right justified. The field width of four is used because the largest product requires three digits; this ensures that no entries will “run together” and look like one value.
- `row` is the control variable for the outer loop; `column` controls the inner loop.
- The inner loop executes ten times on every single iteration of the outer loop. How many times is the statement

```
System.out.printf("%4d", row*column);
```

executed? $10 \times 10 = 100$, one time for every product in the table.

- A newline is printed (by the only `println()` call inside the loops) after the contents of each row is displayed; thus, all the values printed in the inner (`column`) loop appear on the same line.

17.3 Infinite Loops

An infinite loop is a loop that is never exited. Once the program flow enters the loop’s body it cannot escape. Infinite loops are sometimes designed. For example, a long-running server application like a Web server may need to continuously check for incoming queries. This checking can be performed within a loop that runs indefinitely. All

too often for beginning programmers, however, infinite loops are created by accident and represent a logical error in the program.

Intentional infinite loops should be made obvious. For example,

```
while ( true ) {  
    /* Do something forever . . . */  
}
```

The Boolean literal `true` is always true, so it is impossible for the loop's condition to be false. The only ways to exit the loop is via a `break` statement or a `return (§ 4.1)` statement embedded somewhere within its body.

Intentional infinite loops are easy to write correctly. Accidental infinite loops are quite common, but can be puzzling for beginning programmers to diagnose and repair. Consider `FindFactors` (Listing 17.6) that purportedly prints all the integers with their associated factors from 1 to 20.

```
public class FindFactors {  
    public static void main(String[] args) {  
        // List of the factors of the numbers up to 20  
        int n = 1;  
        final int MAX = 20;  
        while ( n <= MAX ) {  
            int factor = 1;  
            System.out.print(n + ": ");  
            while ( factor <= n ) {  
                if ( n % factor == 0 ) {  
                    System.out.print(factor + " ");  
                    factor++;  
                }  
            }  
            System.out.println();  
            n++;  
        }  
    }  
}
```

Listing 17.6: `FindFactors`—an erroneous factoring program

It displays

```
1: 1  
2: 1 2  
3: 1
```

and then “freezes up” or “hangs,” ignoring any user input (except the key sequence **Ctrl C** on most systems which interrupts and terminates the running JVM program). This type of behavior is a frequent symptom of an unintentional infinite loop. The factors of 1 display properly, as do the factors of 2. The first factor of 3 is properly displayed and

then the program hangs. Since the program is short, the problem may be easy to locate. In general, though, the logic can be quite complex. Even in `FindFactors` the debugging task is nontrivial since nested loops are involved. (Can you find and fix the problem in `FindFactors` before reading further?)

In order to avoid infinite loops, we must ensure that the loop exhibits certain properties:

- The loop's condition must not be a tautology (a Boolean expression that can never be false). For example,

```
while ( i >= 1 || i <= 10 ) {
    /* Body omitted */
}
```

is an infinite loop since any value chosen for `i` will satisfy one or both of the two subconditions. Most likely the `&&` operator was intended here so the loop continues until `i` is outside the range 1–10.

In `FindFactors`, the outer loop condition is

```
n <= MAX
```

If `n` is 1 and `MAX` is two, then the condition is false, so this is not a tautology. Checking the inner loop condition:

```
factor <= n
```

we see that if `factor` is 3 and `n` is 2, then the expression is false; therefore, it also is not a tautology.

- The condition of a `while` must be true initially to gain access to its body. The code within the body must modify the state of the program in some way so as to influence the outcome of the condition that is checked at each iteration. This usually means one of the variables used in the condition is modified in the body. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

In `FindFactors`, the outer loop's condition involves the variable `n` and constant `MAX`. `MAX` cannot change, so it is essential that `n` be modified within the loop. Fortunately, the last statement in the body of the outer loop increments `n`. `n` is initially 1 and `MAX` is 20, so unless the circumstances arise to make the inner loop infinite, the outer loop should eventually terminate.

The inner loop's condition involves the variables `n` and `factor`. No statement in the inner loop modifies `n`, so it is imperative that `factor` be modified in the loop. The good news is `factor` is incremented in the body of the inner loop, but the bad news is the increment operation is protected within the body of the `if` statement. The inner loop contains one statement, the `if` statement (the `if` statement in turn has two statements in its body):

```
while ( factor <= n ) {
    if ( n % factor == 0 ) {
        System.out.print(factor + " ");
        factor++;
    }
}
```

If the condition of the `if` is ever false, then the state of the program will not change when the body of the inner loop is executed. This effectively creates an infinite loop. The statement that modifies `factor` must be moved outside of the `if` statement's body:

```
while ( factor <= n ) {  
    if ( n % factor == 0 ) {  
        System.out.print(factor + " ");  
    }  
    factor++;  
}
```

This new version runs correctly.

A debugger can be used to step through a program to see where and why an infinite loop arises. Another common technique is to put print statements in strategic places to examine the values of the variables involved in the loop's control. The original inner loop can be so augmented:

```
while ( factor <= n ) {  
    System.out.println("factor = " + factor + "  n = " + n);  
    if ( n % factor == 0 ) {  
        System.out.print(factor + " ");  
        factor++;  
    }  
}
```

It produces the following output:

```
1: factor = 1  n = 1  
1  
2: factor = 1  n = 2  
1 factor = 2  n = 2  
2  
3: factor = 1  n = 3  
1 factor = 2  n = 3  
factor = 2  n = 3  
factor = 2  n = 3  
factor = 2  n = 3  
factor = 2  n = 3  
factor = 2  n = 3  
.  
.  
.
```

The output demonstrates that once `factor` becomes equal to 2 and `n` becomes equal to 3 the program's execution becomes trapped in the inner loop. Under these conditions:

1. $2 < 3$ is true, so the loop continues and
2. $3 \% 2$ is equal to 1, so the `if` will not increment `factor`.

It is imperative that `factor` be incremented each time through the inner loop; therefore, the statement incrementing `factor` must be moved outside of the `if`'s guarded body.

17.4 Summary

- The `while` allows the execution of code sections to be repeated multiple times.
- The condition of the `while` controls the execution of statements within the `while`'s body.
- The statements within the body of a `while` are executed over and over until the condition of the `while` is false.
- In an infinite loop, the `while`'s condition never becomes false.
- The statements within the `while`'s body must eventually lead to the condition being false; otherwise, the loop will be infinite.
- Infinite loops are rarely intentional and usually accidental.
- An infinite loop can be diagnosed by putting a printing statement inside its body.
- An assignment expression has a value; the expression's value is the same as the expression of the right of the assignment operator.
- Chained assignment allows multiple variables to be assigned the same value in one statement.
- A loop contained within another loop is called a nested loop.
- A loop contained within another loop is called a

17.5 Exercises

1. In `AddUpNonnegatives` (▮17.3) could the condition of the `if` statement have used `>` instead of `>=` and achieved the same results? Why?
2. In `AddUpNonnegatives` (▮17.3) could the condition of the `while` statement have used `>` instead of `>=` and achieved the same results? Why?
3. In `AddUpNonnegatives` (▮17.3) what would happen if the statement assigning `scan` is moved into the loop? Is moving the assignment into the loop a good or bad thing to do? Why?
4. In the following code fragment:

```
int a = 0;
while (a < 100) {
    int b = 0;
    while (b < 55) {
        System.out.println("*");
    }
}
```

how many asterisks are printed?

5. Modify `TimesTable` (▮17.5) so that instead of using a `main()` method it uses a `show()` method to print the table. The `show()` method should accept an integer parameter representing the size of the table: 10 prints a 10×10 table, 15 prints a 15×15 table, etc. Accept any integer values from 1 to 18. Be sure everything lines up correctly, and the table looks attractive.

Chapter 18

Examples using Iteration

Some sophisticated algorithms can be implemented as Java programs now that we are armed with `if` and `while` statements. This chapter provides a number of examples that show off the power of conditional execution and iteration.

18.1 Example 1: Drawing a Tree

Suppose such a tree must be drawn, but its height is provided by the user. `StarTree` (Figure 18.1) provides the necessary functionality.

```
import java.util.Scanner;

public class StarTree {
    public static void main(String[] args) {
        Scanner kbd = new Scanner(System.in);
        int height;    // Height of tree
        System.out.print("Enter height of tree: ");
        height = kbd.nextInt(); // Get height from user
        int row = 0;      // First row, from the top, to draw
        while ( row < height ) { // Draw one row for every unit of height
            // Print leading spaces
            int count = 0;
            while ( count < height - row ) {
                System.out.print(" ");
                count++;
            }
            // Print out stars, twice the current row plus one:
            // - number of stars on left side of tree = current row value
            // - exactly one star in the center of tree
            // - number of stars on right side of tree = current row value
            count = 0;
            while ( count < 2*row + 1 ) {
                System.out.print("*");
            }
        }
    }
}
```

```

        count++;
    }
    // Move cursor down to next line
    System.out.println();
    // Change to the next row
    row++;
}
}
}

```

Listing 18.1: StarTree—Draw a tree of asterisks given a user supplied height

When StarTree is run and the user enters 7, the output is:

```

Enter height of tree: 7
  *
 ***
*****
*****
*****
*****
*****

```

StarTree uses two `while` loops nested within a `while` loop. The outer `while` loop is responsible for drawing one row of the tree each time its body is executed:

- As long as the user enters a value greater than zero, the body of the outer `while` loop will be executed; if the user enters zero or less, the program terminates and does nothing.
- The last statement in the body of the outer `while`,

```
row++;
```

ensures that the variable `row` increases by one each time through the loop; therefore, it eventually will equal `height` (since it initially had to be less than `height` to enter the loop), and the loop will terminate. There is no possibility of an infinite loop here.

- The body consists of more than one statement; therefore, it must be enclosed within curly braces. Whenever a group of statements is enclosed within curly braces a *block* is formed. Any variable declared within a block is local to that block. Its scope is from its point of declaration to the end of the block. For example, the variables `height` and `row` are declared in the block that is `main()`'s body; thus, they are local to `main()`. The variable `count` is declared within the block that is the body of the `while` statement; therefore, `count` is local to the `while` statement. An attempt to use `count` *after* the `while` statement outside its body would be an error.

The two inner loops play distinct roles:

- The first inner loop prints spaces. The number of spaces printed is equal to the height of the tree the first time through the outer loop and decreases each iteration. This is the correct behavior since each succeeding row moving down contains fewer leading spaces but more asterisks.

- The second inner loop prints the row of asterisks that make up the tree. The first time through the outer loop, row is zero, so no left side asterisks are printed, one central asterisk is printed (the top of the tree), and no right side asterisks are printed. Each time through the loop the number of left-hand and right-hand stars to print both increase by one and the same central asterisk is printed; therefore, the tree grows one wider on each side each line moving down. Observe how the $2 \times \text{row} + 1$ value expresses the needed number of asterisks perfectly.

18.2 Example 2: Prime Number Determination

A *prime number* is an integer greater than one whose only factors (also called divisors) are one and itself. For example, 29 is a prime number (only 1 and 29 divide into it with no remainder), but 28 is not (2 is a factor of 28). Prime numbers were once merely an intellectual curiosity of mathematicians, but now they play an important role in cryptography.

The task is to write a program that displays all the prime numbers up to a value entered by the user. `PrintPrimes` (Figure 18.2) provides one solution.

```
import java.util.Scanner;

public class PrintPrimes {
    static boolean isPrime(int n) {
        for ( int trialFactor = 2; trialFactor <= Math.sqrt(n);
              trialFactor++ ) {
            if ( n % trialFactor == 0 ) { // Is trialFactor a factor?
                return false; // Yes, return right away
            }
        }
        return true; // Tried them all, must be prime
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        int value = 2; // Smallest prime number
        while ( value <= maxValue ) {
            // See if value is prime
            if (isPrime(value)) {
                System.out.print(value + " "); // Display the prime number
            }
            value++; // Try the next potential prime number
        }
        System.out.println(); // Move cursor down to next line
    }
}
```

Listing 18.2: `PrintPrimes`—Prime number generator

Figure 18.2, with an input of 90, produces:

```
Display primes up to what value? 90
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
```

The logic of `PrintPrimes` is a little more complex than that of `StarTree` (Figure 18.1). The user provides a value for `maxValue`. The main loop (outer `while`) iterates over all the values from two to `maxValue`.

- Two new variables, local to the body of the outer loop, are introduced: `trialFactor` and `isPrime`. `isPrime` is initialized to `true`, meaning `value` is assumed to be prime unless our tests prove otherwise. `trialFactor` takes on all the values from two to `value - 1` in the inner loop:

```
int trialFactor = 2;
while ( trialFactor < value - 1 ) {
    if ( value % trialFactor == 0 ) { // Is trialFactor a factor?
        isPrime = false;
        break; // No need to continue, we found a factor
    }
    trialFactor++; // Try next factor
}
```

If any of the values of `trialFactor` is determined to actually be a factor of `value`, then `isPrime` is set to `false`, and the loop is prematurely exited. If the loop continues to completion, `isPrime` will never be set to `false`, which means no factors were found and `value` is indeed prime.

- The `if` statement after the inner loop:

```
if ( isPrime ) {
    System.out.print(value + " "); // Display the prime number
}
```

simply checks the status of `isPrime`. If `isPrime` is `true`, then `value` must be prime, so `value` (along with an extra space so numbers printed later will not run together) will be printed.

Some important questions can be asked.

- If the user enters a 2, will it be printed?** `maxValue = value = 2`, so the condition of the outer loop

```
value <= maxValue
```

is true, since $2 \leq 2$. `isPrime` is set to `true`, but the condition of the inner loop

```
trialFactor < value - 1
```

is not true ($2 \not< 2 - 1$). Thus, the inner loop is skipped, `isPrime` is not changed from `true`, and 2 is printed.

- Is the inner loop guaranteed to always terminate?** In order to enter the body of the inner loop, `trialFactor < value - 1`. `value` does not change anywhere in the loop. `trialFactor` is not modified anywhere in the `if` statement, and it is incremented immediately after the `if` statement. Therefore, eventually `trialFactor` will equal `value - 1`, and the loop will terminate.

3. **Is the outer loop guaranteed to always terminate?** In order to enter the body of the outer loop, $\text{value} \leq \text{maxValue}$. maxValue does not change anywhere in the loop. value is increased in the last statement within the body of the outer loop, and since the inner loop is guaranteed to terminate as shown in the previous answer, eventually value will exceed maxValue and the loop will end.

The logic of the inner `while` can be rearranged slightly to avoid the `break` statement. The current version is:

```
boolean isPrime = true; // Assume no factors unless we find one
while ( trialFactor < value - 1 ) {
    if ( value % trialFactor == 0 ) { // Is trialFactor a factor?
        isPrime = false;
        break; // No need to continue, we found a factor
    }
    trialFactor++; // Try next factor
}
```

It can be rewritten as:

```
boolean isPrime = true; // Assume no factors unless we find one
while ( isPrime && trialFactor < value - 1 ) {
    isPrime = !(value % trialFactor == 0);
    trialFactor++; // Try next factor
}
```

This version without the `break` introduces a slightly more complicated condition for the `while` but removes the `if` statement within its body. Profiling reveals that neither loop structure offers a performance advantage over the other.

The performance can be enhanced, however, with a modification to the algorithm. The algorithm can be made more efficient. For example, if a number n is prime, `PrintPrimes` can only certify that n is prime after testing all the integers in the range $2 \dots n-1$. For large n , this could take some time. A simple optimization is based on the fact that if n has no factors in the range $2 \dots \sqrt{n}$, then n must be prime. Fortunately, a standard class named `Math` provides a square root method. To use this square root method, replace the condition of the inner loop of `PrintPrimes` with

```
// Was: while ( trialFactor < value - 1 ) {
// New version:
while ( trialFactor <= Math.sqrt(value) ) {
```

The `Math.sqrt()` method computes the square root of the double argument passed to it. The result returned is a double `value`. Automatic widening converts the `int` `value` to a double.

Does this optimization make any difference? Since computers are so fast, perhaps this minor change makes little or no difference in the speed of the program. A technique known as *profiling* can answer this question. The simplest form of profiling involves timing a section of code to see how long it takes to execute. The original code is modified to permit this timing. The act of modifying source code to derive information beyond what the original code is intended to provide is called *instrumenting* the code. Code can be instrumented for reasons other than performance testing.

`InstrumentedPrimes` (▮ 18.3) is the instrumented version of `PrintPrimes` (▮ 18.2).

```
import java.util.Scanner;
```

```

public class InstrumentedPrimes {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        int value = 2; // Smallest prime number
        long startTime = System.currentTimeMillis();
        while ( value <= maxValue ) {
            // See if value is prime
            int trialFactor = 2;
            boolean isPrime = true; // Assume no factors unless we find one
            // while ( trialFactor <= Math.sqrt(value) ) {
            while ( trialFactor < value - 1 ) {
                if ( value % trialFactor == 0 ) { // Is trialFactor a factor?
                    isPrime = false;
                    break; // No need to continue, we found a factor
                }
                trialFactor++; // Try next factor
            }
            // if ( isPrime ) {
            //     System.out.print(value + " "); // Display the prime number
            // }
            value++; // Try the next potential prime number
        }
        // System.out.println(); // Move cursor down to next line
        System.out.println("Elapsed time: " +
                           (System.currentTimeMillis() - startTime) +
                           " msec");
    }
}

```

Listing 18.3: InstrumentedPrimes—Instrumented prime number generator

The original version was modified as follows:

- The new inner while appears that uses `Math.sqrt()` is commented out, so the original condition is still checked. This configuration is used to time the original version. To test the new version, *uncomment* the new version and comment out the original version.
- The method `System.currentTimeMillis()` gets information from the operating system's clock to determine the number of milliseconds since midnight January 1, 1970. Calling it twice and comparing the two results indicates the elapsed time in milliseconds between the two calls.
- The original output statements (the `print()` and `println()` calls) are commented out. We wish to compare the raw speed of the two versions, and I/O tends to slow things down considerably. If we try large values (like 10,000) the slowdown from displaying all the primes would somewhat obscure the speed difference between the two programs. We're testing raw processor speed, not the speed of the I/O devices.

- The new print statement displays the difference in times (elapsed time) after the outer loop has terminated.

Three runs of the instrumented original version on one computer system produce¹:

```
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49360 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49359 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 49360 msec
```

Three runs of the instrumented square root version produce:

```
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 790 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 799 msec
#>java PrintPrimes
Display primes up to what value? 100000
Elapsed time: 792 msec
```

All the primes up to 100,000 are found during each run. The difference is dramatic. The original version averages a little over 49 seconds to find all the prime numbers; the optimized version takes less than one second! The change was minor, but the effect was major. This illustrates the value of a quality algorithm. A good algorithm is both correct *and* efficient.

18.3 Example 3: Digital Timer Display

We can put the `System.currentTimeMillis()` method to good use in implementing a digital timer. `DigitalTimer` (Figure 18.4) implements a digital timer using loops, conditional statements and the `System.currentTimeMillis()` method.

```
public class DigitalTimer {
    private static long hours = 0;
    private static long minutes = 0;
    private static long seconds = 0;
    public static void main(String[] args) {
```

¹Note: if you run this on a multi-user system with a number of people logged on and doing work, the results may differ more widely between each run of the same version of the program. The elapsed time computed from the `System.currentTimeMillis()` calls indicates the real-time elapsed time, not the time dedicated exclusively to your Java process.

```

// Some conversions from milliseconds
final long SECONDS = 1000, // 1000 msec = 1 sec
           MINUTES = 60 * SECONDS, // 60 seconds = 1 minute
           HOURS = 60 * MINUTES, // 60 minutes = 1 hour
           _24_HOURS = 24 * HOURS; // 24 hours = 24 hours

// Record starting time
long start = System.currentTimeMillis();
// Elapsed time
long elapsed = System.currentTimeMillis() - start,
previousElapsed = elapsed;

// Counts up to 24 hours, then stops
while ( elapsed < _24_HOURS ) {
    // Update the display only every second
    if ( elapsed - previousElapsed >= 1000 ) {
        // Remember when we last updated the display
        previousElapsed = elapsed;
        // Compute hours
        hours = elapsed/HOURS;
        // Remove the hours from elapsed
        elapsed %= HOURS;
        // Compute minutes
        minutes = elapsed/MINUTES;
        // Remove the minutes from elapsed
        elapsed %= MINUTES;
        // Compute seconds
        seconds = elapsed/SECONDS;
        // Display results with leading zeroes
        System.out.printf("%02d:%02d:%02d\n",
                           hours, minutes, seconds);
    }
    // Update time: number of milliseconds since starting
    elapsed = System.currentTimeMillis() - start;
}
}
}

```

Listing 18.4: DigitalTimer—a digital timer program that accurately keeps time

The operating system has an accurate real-world clock (as accurate as the hardware clock that the OS uses, at least). We can continuously get the current time to measure the time difference from when the timer began. The absolute number of milliseconds since the timer was started then can be displayed in an hours:minutes:seconds format.

In DigitalTimer:

- The timer begins at 00:00:00 and counts up to 23:59:59. When it reaches its maximum value it stops and the program terminates.

- Three variables keep track of the hours, minutes, and seconds that the timer displays. The variables cannot be displayed directly as is; otherwise, the display 12:2:4 will result when 12:02:04 is desired. We use the `%02d` control code in `System.out.printf()` to print two-digit numbers with leading zeroes, if necessary.
- The key variable is `elapsed`, which keeps track of milliseconds since starting the timer. It is updated at each time through the loop.
- Four constants are defined to simplify the calculations in the code. Each represents the number of milliseconds in the time interval specified by its name. Observe how a previously defined constant can be used in the definition of a new constant.
- Each time through its loop `DigitalTimer` checks the total elapsed time from the start.
- The body of the loop can be executed many times each second. If the display were updated each time through the loop, most of the time the new display would be unchanged.
- The appropriate hours, minutes, and seconds are derived from the milliseconds elapsed since the start. We start with the highest time unit, hours, and work toward the lowest unit, seconds. Observe that the process is regular:
 - divide `elapsed` by the time unit to get the number of those time units expressed in the milliseconds stored in `elapsed` and
 - use the modulus operator with the time unit to get the remainder of milliseconds left over after removing the milliseconds represented by that time unit—this remainder is used to compute the lower time units.

18.4 A Mutable List

Recall `List` (§16.7), an abstract class that defines the interface of list objects. In our original implementation in §16.2 our lists were immutable, meaning an existing list could not be changed. Appending to an existing list produced a copy of the list with the new element tacked onto the end. Immutable objects—like Java’s `String` objects—offer some inherent advantages over mutable objects. In general it is easier to reason about the correctness of programs when objects are immutable. Since an immutable object’s state cannot change, when the object’s value is determined at one point in the program’s execution it is guaranteed to have that same value at any point later during the program’s execution.

Immutability does have its price, however. Object creation is a relatively expensive operation compared to say, assigning a number to a numeric variable or testing a condition. Making a copy of an object when we need a modified form of it is inefficient if we never intend to use the original version again. In the case of our earlier list implementation, suppose we have a list containing 10,000 elements, and we wish to add a new element to the end:

```
// lst is a NonemptyList containing 10,000 elements.
lst = lst.append("Betty");
```

The right-hand side of the assignment creates a new list and copies all the elements from `lst` plus "Betty" to the new list object. The assignment operator directs the reference `lst` to refer to this newly created list. Appending our new element thus results in the creation of 10,001 objects! A more efficient approach would create one new object ("Betty") and somehow link it into the existing list.

We can begin with the same abstract class, `List` (§16.7), and create a mutable list object, `MutableList` (§18.6):

```

public class ListNode {
    public Object element;
    public ListNode next;
    public ListNode(Object elem) {
        element = elem;
        next = null;
    }
}

```

Listing 18.5: ListNode—a primitive building block for mutable lists

```

public class MutableList extends List {

    private ListNode first; // First node in the list (null if empty)
    private ListNode last;  // Last node in the list (null if empty)

    public MutableList() {
        first = last = null; // List initially empty
    }

    // Counts the number of elements in the list
    public int length() {
        int len = 0;
        ListNode cursor = first; // Cursor steps through list;
                                // set it to the beginning
        while (cursor != null) { // While we have not reached the
                                // of the list . . .
            len++; // Count the current element
            cursor = cursor.next; // Move to the next element
        }
        return len;
    }

    // Adds an element to the end of the list without creating a copy
    // of the list.
    public List append(Object newElement) {
        // Make node for new element
        ListNode newNode = new ListNode(newElement);
        if (first == null) { // Empty list
            first = last = newNode;
        } else {
            last.next = newNode; // Add to current end of list
            last = newNode; // Update the end of list to the
                            // to the new element
        }
        return this; // Return current list to comply
                    // with superclass requirements
    }

    protected String toStringHelper() {

```

```

        String result = " ";           // Empty, until determined
                                       // otherwise
        ListNode cursor = first;       // Consider first element, if it
                                       // exists
        while (cursor != null) {       // Continue until we reach the end
                                       // of the list
            result += cursor.element + " ";
            // Move to next element in the list
            cursor = cursor.next;
        }
        return result;
    }
}

```

Listing 18.6: MutableList—a class for building mutable list objects

In addition to mutability, our new list class uses iteration exclusively instead of recursion. Iteration is more efficient than recursion, because a method call is relatively expensive compared to a non-method call statement (such as an assignment or conditional check). Whenever a method is called, some information needs to be stored in memory:

- parameters, if applicable (separate copies for each recursive invocation)
- local variables, if applicable (separate copies for each recursive invocation)
- location of the call (return address), so execution can return to the correct position within the code when the method is finished with its invocation

Storing this information takes time and, of course, extra memory.

MutableListTest (Figure 18.7) tests the `append()` and `toString()` methods of our mutable list:

```

import junit.framework.TestCase;

public class MutableListTest extends TestCase {

    public void testAppendMethod() {
        List list = new MutableList();
        assertEquals("[ ]", list.toString());
        assertEquals(0, list.length());
        list = list.append(22);
        //System.out.println(list);
        assertEquals("[ 22 ]", list.toString());
        assertEquals(1, list.length());
        list = list.append("Fred");
        assertEquals("[ 22 Fred ]", list.toString());
        assertEquals(2, list.length());
        list = list.append(2.2);
        assertEquals("[ 22 Fred 2.2 ]", list.toString());
        assertEquals(3, list.length());
    }
}

```

```
list = list.append("Wilma");
assertEquals("[ 22 Fred 2.2 Wilma ]", list.toString());
assertEquals(4, list.length());
    }
}
```

Listing 18.7: MutableListTest—testing our mutable list

18.5 Summary

- Iteration is a powerful mechanism and can be used to solve many interesting problems.
- A block is any section of source code enclosed within curly braces.
- A variable declared within a block is local to that block.
- Complex iteration using nested loops mixed with conditional statements can be difficult to do correctly.
- Sometimes simple optimizations can speed up considerably the execution of loops.

18.6 Exercises

1. Add item here

Chapter 19

Other Conditional and Iterative Statements

The `if/else` and `while` statements are flexible enough to implement the logic of any algorithm we might wish to implement, but Java provides some additional conditional and iterative statements that are more convenient to use in some circumstances. These additional statements include

- `switch`: an alternative to some multi-way `if/else` statements
- conditional operator: an expression that exhibits the behavior of an `if/else` statement
- `do/while`: a loop that checks its condition after its body is executed
- `for`: a loop convenient for counting

19.1 The `switch` Statement

The `switch` statement provides a convenient alternative for some multi-way `if/else` statements like the one in `RestyledDigitToWord` (Figure 6.7). The general form of a `switch` is:

```
switch ( integral expression ) {  
    case integral constant 1 :  
        statement(s)  
        break;  
    case integral constant 2 :  
        statement(s)  
        break;  
    case integral constant 3 :  
        statement(s)  
        break;  
        .  
        .  
        .  
    case integral constant n :  
        statement(s)  
        break;  
    default:  
        statement(s)  
        break;  
}
```

- The reserved word `switch` identifies a switch statement.
- The expression, contained in parentheses, must evaluate to an integral value. Any integer type, characters, and Boolean expressions are acceptable. Floating point expressions are forbidden.
- The body of the switch is enclosed by curly braces, which are required.
- Each `case` label is followed by an *integral constant*. This constant can be either a literal value or a final symbolic value. In particular, non-final variables and other expressions are expressly forbidden. If the `case` label matches the `switch`'s expression, then the statements that follow that label are executed. The statements and `break` statement that follow each `case` label are optional. One way to execute one set of statements for more than one `case` label is to provide empty statements for one or more of the labels, as in:

```
switch ( inKey ) {  
    case 'p':  
    case 'P':  
        System.out.println("Executing print");  
        break;  
    case 'q':  
    case 'Q':  
        done = true;  
}
```

```
break;
}
```

Here either an upper- or lowercase *P* result in the same action; either an upper- or lowercase *Q* sets the done Boolean variable. The break statement is optional. When a case label is matched, the statements that follow are executed until a break statement is encountered. The control flow then transfers out of the body of the switch. (In this way, the break within a switch works just like a break within a loop: the rest of the body of the statement is skipped and program execution resumes at the next statement following the body.) A missing break (a common error, when its omission is not intentional) causes the statements of the succeeding case label to be executed. The process continues until a break is encountered or the end of the body is reached.

- The default label is matched if none of the case labels match. It serves as a “catch all” like the final else in a multi-way if/else statement. The default label is optional. If it is missing and none of the case labels match the expression, then no statements in the switch’s body are executed.

SwitchDigitToWord (Figure 19.1) shows what RestyledDigitToWord (Figure 6.7) would look like with a switch statement instead of the multi-way if/else statement.

```
import java.util.Scanner;

public class SwitchDigitToWord {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int value;
        System.out.println("Please enter an integer in the range 0...5: ");
        value = scanner.nextInt();
        if ( value < 0 ) {
            System.out.println("Too small");
        } else {
            switch ( value ) {
                case 0:
                    System.out.println("zero");
                    break;
                case 1:
                    System.out.println("one");
                    break;
                case 2:
                    System.out.println("two");
                    break;
                case 3:
                    System.out.println("three");
                    break;
                case 4:
                    System.out.println("four");
                    break;
                case 5:
                    System.out.println("five");
                    break;
                default:
```

```

        System.out.println("Too large");
    }
}
}
}

```

Listing 19.1: SwitchDigitToWord—switch version of multi-way if

The switch statement has two restrictions that make it less general than the multi-way if/else:

1. The switch argument must be an integral expression.
2. Case labels must be constant integral values. Integral literals and constants are acceptable. Variables or expressions are *not* allowed.

To illustrate these restrictions, consider the following if/else statement that translates easily to an equivalent switch statement:

```

if ( x == 1 ) {
    // Do 1 stuff here . . .
} else if ( x == 2 ) {
    // Do 2 stuff here . . .
} else if ( x == 3 ) {
    // Do 3 stuff here . . .
}

```

The corresponding switch statement is:

```

switch ( x ) {
    case 1:
        // Do 1 stuff here . . .
        break;
    case 2:
        // Do 2 stuff here . . .
        break;
    case 3:
        // Do 3 stuff here . . .
        break;
}

```

Now consider the following if/else:

```

if ( x == y ) {
    // Do "y" stuff here . . .
} else if ( x > 2 ) {
    // Do "> 2" stuff here . . .
} else if ( x == 3 ) {
    // Do 3 stuff here . . .
}

```

This code cannot be easily translated into a `switch` statement. The variable `y` cannot be used as a `case` label. The second choice checks for an inequality instead of an exact match, so direct translation to a `case` label is impossible.

As a consequence of the `switch` statement's restrictions, the compiler produces more efficient code for a `switch` than for an equivalent `if/else`. If a choice must be made from one of several or more options, and the `switch` statement can be used, then the `switch` statement will likely be faster than the corresponding multi-way `if/else`.

19.2 The Conditional Operator

As purely a syntactical convenience, Java provides an alternative to the `if/else` construct called the *conditional operator*. It has limited application but is convenient nonetheless. The following section of code assigns either the result of a division or a default value acceptable to the application if a division by zero would result:

```
// Assign a value to x:
if ( z != 0 ) {
    x = y/z;
} else {
    x = 0;
}
```

This code has two assignment statements, but only one is executed at any given time. The conditional operator makes for a simpler statement:

```
// Assign a value to x:
x = ( z != 0 ) ? y/z : 0;
```

The general form of a conditional expression is:

$$\textit{condition} \ ? \ \textit{expression}_1 \ : \ \textit{expression}_2$$

- *condition* is a normal Boolean expression that might appear in an `if` statement. Parentheses around the condition are not required but should be used to improve the readability.
- *expression*₁ the overall value of the conditional expression if the condition is true.
- *expression*₂ the overall value of the conditional expression if the condition is false.

The conditional operator uses two symbols (`?` and `:`) and three operands. Since it has three operands it is classified as a *ternary* operator (Java's only one). Both *expression*₁ and *expression*₂ must be assignment compatible; for example, it would be illegal for one expression to be an `int` and the other to be `boolean`. The overall type of a conditional expression is the type of the more dominant of *expression*₁ and *expression*₂. The conditional expression can be used anywhere an expression can be used. It is not a statement itself; it is used within a statement.

As another example, the *absolute value* of a number is defined in mathematics by the following formula:

$$|n| = \begin{cases} n, & \text{when } n \geq 0 \\ -n, & \text{when } n < 0 \end{cases}$$

In other words, the absolute value of a positive number or zero is the same as that number; the absolute value of a negative number is the additive inverse (negative of) of that number. The following Java expression represents the *absolute value* of the variable `n`:

`(n < 0) ? -n : n`

Some argue that the conditional operator is cryptic, and thus its use reduces a program's readability. To seasoned Java programmers it is quite understandable, but it is actually used sparingly because of its very specific nature.

19.3 The do/while Statement

The `while` statement (Section 17.1) checks its condition before its body is executed; thus, it is a *top-checking* loop. Its body is not executed if its condition is initially false. At times, this structure is inconvenient. Consider `GoodInputOnly` (Figure 19.2).

```
import javax.swing.JOptionPane;

public class GoodInputOnly {
    public static void main() {
        int inValue = -1;
        while ( inValue < 0 || inValue > 10 ) {
            // Insist on values in the range 0...10
            inValue = Integer.parseInt
                (JOptionPane.showInputDialog
                 ("Enter integer in range 0...10: "));
        }
        // inValue at this point is guaranteed to be within range
        JOptionPane.showMessageDialog(null, "Legal value entered was "
                                     + inValue);
    }
}
```

Listing 19.2: `GoodInputOnly`—Insist the user enter a good value

The loop in `GoodInputOnly` traps the user until he provides a good value. Here's how it works:

- The initialization of `inValue` to `-1` ensures the condition of the `while` will be true, and, thus, the body of the loop will be entered.
- The condition of the `while` specifies a set that includes all values that are *not* in the desired range. `inValue` is initially in this set, so the loop is entered.
- The user does not get a chance to enter a value until program execution is inside the loop.
- The only way the loop can be exited is if the user enters a value that violates the condition—precisely a value in the desired range.

The initialization before the loop check is somewhat artificial. It is there only to ensure entry into the loop. It seems unnatural to check for a valid value *before* the user gets a chance to enter it. A loop that checks its condition after its body is executed at least once would be more appropriate. The `do/while` is *bottom-checking* loop that behaves exactly in this manner. Its flowchart is shown in Figure 19.1.

The `do/while` statement has the general form:

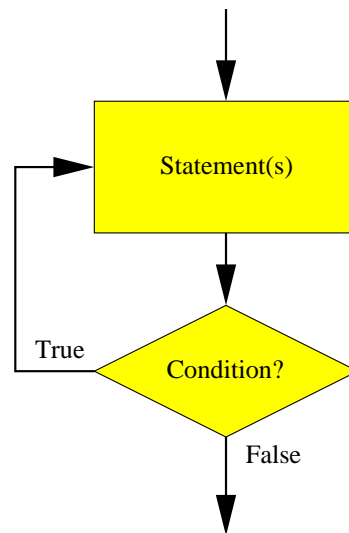


Figure 19.1: Execution flow in a do/while statement

```

do
    body
while ( condition );
  
```

- The reserved words `do` and `while` identify a do/while statement. The `do` and `while` keywords delimit the loop's body, but curly braces are still required if the body consists of more than one statement.
- The condition is associated with the `while` at the end of the loop. The condition must be enclosed within parentheses.
- The body is like the body of the `while` loop.

BetterInputOnly (Figure 19.3) uses a do/while to erase the criticisms of GoodInputOnly (Figure 19.2).

```

import javax.swing.JOptionPane;

public class BetterInputOnly {
    public static void main() {
        int inValue;
        do {
            // Insist on values in the range 0...10
            inValue = Integer.parseInt
                (JOptionPane.showInputDialog
                    ("Enter integer in range 0...10: "));
        } while ( inValue < 0 || inValue > 10 );
        // inValue at this point is guaranteed to be within range
        JOptionPane.showMessageDialog(null, "Legal value entered was "
            + inValue);
    }
}
  
```

Listing 19.3: BetterInputOnly—GoodInputOnly (Figure 19.2) using a do/while loop

The body of a do/while statement, unlike the while statement, is guaranteed to execute at least once. This behavior is convenient at times as BetterInputOnly shows.

We can use BetterInputOnly as a starting point for a general-purpose reusable class, IntRange (Figure 19.4).

```
import javax.swing.JOptionPane;

// Restricts the user to entering a restricted range of integer
// values
public class IntRange {
    public static int get(int low, int high) {
        int inValue;
        do {
            // Insist on values in the range low...high
            inValue = Integer.parseInt
                (JOptionPane.showInputDialog
                 ("Enter integer in range " + low + "... " + high));
        } while ( inValue < low || inValue > high );
        // inValue at this point is guaranteed to be within range
        return inValue;
    }
}
```

Listing 19.4: IntRange—a useful reusable input method

The break and continue statements can be used in the body of a do/while statement. Like with the while statement, break causes immediate loop termination (any remaining statements within the body are skipped), and continue causes the remainder of the body to be skipped and the condition is immediately checked to see if the loop should continue or be terminated.

19.4 The for Statement

Recall IterativeCountToFive (Figure 17.2) from Section 17.1, reproduced here.

It simply counts from one to five. Counting is a frequent activity performed by computer programs. Certain program elements are required in order for any program to count:

- A variable must be used to keep track of the count; count is the aptly named counter variable.
- The counter variable must be given an initial value, 1.
- The variable must be modified (usually incremented) as the program counts. The statement

```
count = count + 1;
```


increments count.

- A way must be provided to determine if the count has completed. The condition of the `while` controls the extent of the count.

Java provides a specialized loop that packages these four programming elements into one convenient statement. Called the `for` statement, its general form is

`for (initialization ; condition ; modification)`
`body`

- The reserved word `for` identifies a `for` statement.
- The header, contained in parentheses, contains three parts, each separated by semicolons:
 1. **Initialization.** The initialization part assigns an initial value to the loop variable. The loop variable may be declared here as well; if it is declared here, then its scope is limited to the `for` statement. The initialization part is performed one time.
 2. **Condition.** The condition part is a Boolean expression, just like the condition of `while` and `do/while`. The condition is checked each time before the body is executed.
 3. **Modification.** The modification part changes the loop variable. The change should be such that the condition will eventually become false so the loop will terminate. The modification is performed during each iteration *after* the body is executed.
- The body is like the body of any other loop.

With a `while` loop, these four counting components (variable declaration, initialization, condition, and modification) can be scattered throughout the method. With a `for` loop, the programmer can determine all the important information about the loop's control by looking at one statement. Figure 19.2 shows the control flow within a `for` statement.

`ForCounter` (Figure 19.5) uses a `for` loop to do the work of `IterativeCountToFive` (Figure 17.2).

```
public class ForCounter {
    public static void main(String[] args) {
        for ( int count = 1; count <= 5; count++ ) {
            System.out.println(count);
        }
    }
}
```

Listing 19.5: `ForCounter`—`IterativeCountToFive` (Figure 17.2) using a `for` statement in place of the `while`

`TimesTable` (Figure 17.5) that prints a multiplication table is better written with nested `for` statements as `BetterTimesTable` (Figure 19.6) shows.

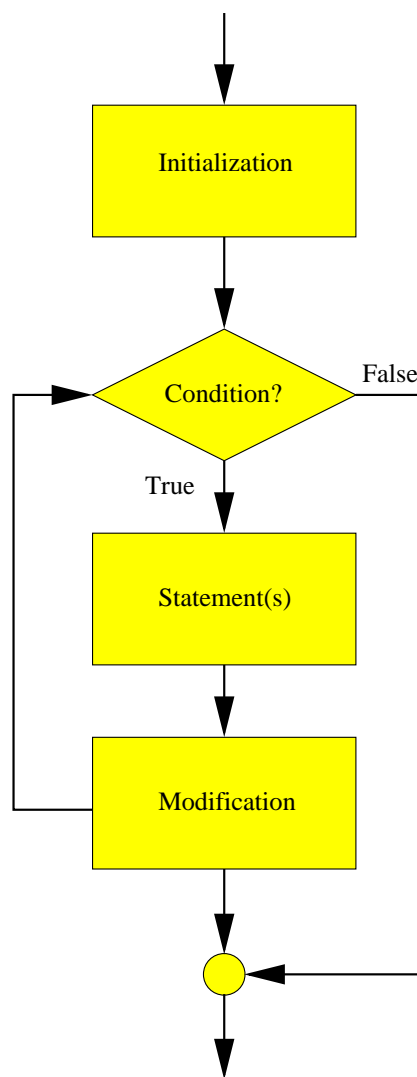


Figure 19.2: Execution flow in a for statement

```

public class BetterTimesTable {
    public static void main(String[] args) {
        // Print a multiplication table to 10 x 10
        // Print column heading
        System.out.println("      1   2   3   4   5   6   7   8   9  10");
        System.out.println("  +-----");
        for ( int row = 1; row <= 10; row++ ) {    // Table has ten rows.
            System.out.printf("%3d|", row);    // Print heading for this row.
            for ( int column = 1; column <= 10; column++ ) {
                System.out.printf("%4d", row*column);
            }
            System.out.println();    // Move cursor to next row
        }
    }
}

```

Listing 19.6: BetterTimesTable—Prints a multiplication table using for statements

A `for` loop is ideal for stepping through the rows and columns. The information about the control of both loops is now packaged in the respective `for` statements instead of being spread out in various places in `main()`. In the `while` version, it is easy for the programmer to forget to update one or both of the counter variables (`row` and/or `column`). The `for` makes it harder for the programmer to forget the loop variable update, since it is done right up front in the `for` statement header.

It is considered bad programming practice to do either of the following in a `for` statement:

- **Modify the loop control variable within the body of the loop.** If the loop variable is modified within the body, then the logic of the loop's control is no longer completely isolated to the `for` statement's header. The programmer must look elsewhere within the statement to understand completely how the loop works.
- **Prematurely exit the loop with a `break`.** This action also violates the concept of keeping all the loop control logic in one place (the `for`'s header).

The language allows both of these practices, but experience shows that it is best to avoid them. If it seems necessary to violate this advice, consider using a different kind of loop (`while` or `do/while`) that does not imply the same degree of control regularity implied by the `for` loop.

`ForPrintPrimes` (▮ 19.7) is a rewrite of `PrintPrimes` (▮ 18.2) that replaces its `while` loops with `for` loops.

```
import java.util.Scanner;

public class ForPrintPrimes {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int maxValue;
        System.out.print("Display primes up to what value? ");
        maxValue = scan.nextInt();
        for ( int value = 2; value <= maxValue; value++ ) {
            // See if value is prime
            int trialFactor = 2;
            boolean isPrime = true; // Assume no factors unless we find one
            for ( trialFactor = 2; isPrime && trialFactor < value - 1;
                  trialFactor++ ) {
                if ( value % trialFactor == 0 ) { // Is trialFactor a factor?
                    isPrime = false;
                    break; // No need to continue, we found a factor
                }
            }
            if ( isPrime ) {
                System.out.print(value + " "); // Display the prime number
            }
        }
        System.out.println(); // Move cursor down to next line
    }
}
```

Listing 19.7: `ForPrintPrimes`—`PrintPrimes` using `for` loops

The conditional in the `for` loop can be any legal Boolean expression. The logical *and* (`&&`), *or* (`||`), and *not* (`!`) operators can be used to create complex Boolean expressions, if necessary. The modification part of the `for` loop is not limited to simple arithmetic and can be quite elaborate. For example:

```
for ( double d = 1000; d >= 1; d = Math.sqrt(d) ) {
    /* Body goes here */
}
```

Here `d` is reassigned by a method call. The following loop is controlled entirely by user input (`scan` is a `Scanner` object):

```
for ( int i = scan.nextInt(); i != 999; i = scan.nextInt() ) {
    /* Body goes here */
}
```

While the `for` statement supports such complex headers, simpler is usually better. Ordinarily the `for` loop should manage just one control variable, and the initialization, condition, and modification parts should be straightforward. If a particular programming situation warrants a particularly complicated `for` construction, consider using another kind of loop.

Any or all of the parts of the `for` statement (initialization, condition, modification, and body) may be omitted:

- **Initialization.** If the initialization is missing, as in

```
for ( ; i < 10; i++ ) {
    /* Body goes here */
}
```

then no initialization is performed by the `for` loop, and it must be done elsewhere.

- **Condition.** If the condition is missing, as in

```
for ( int i = 0; ; i++ ) {
    /* Body goes here */
}
```

then the condition is `true` by default. A `break` must appear in the body unless an infinite loop is intended.

- **Modification.** If the modification is missing, as in

```
for ( int i = 0; i < 10; ) {
    /* Body goes here */
}
```

then the `for` performs no automatic modification; the modification must be done by a statement in the body to avoid an infinite loop.

- **Body.** If the body is missing, as in

```
for ( int i = 0; i < 10; i++ ) {}
```

or

```
for ( int i = 0; i < 10; i++ );
```

then an empty loop results. This can be used for a nonportable delay (slower computers will delay longer than faster computers), but some compilers may detect that such code has no functional effect and “optimize” away such an empty loop.

While the `for` statement supports the omission of parts of its header, such constructs should be avoided. The `for` loop’s strength lies in the ability for the programmer to see all the aspects of the loop’s control in one place. If some of these control responsibilities are to be handled elsewhere (not in the `for`’s header) then consider using another kind of loop.

Programmers usually select a simple name for the control variable of a `for` statement. Recall that variable names should be well chosen to reflect the meaning of their use within the program. It may come as a surprise that `i` is probably the most common name used for an integer control variable. This has its roots in mathematics where variables such as *i*, *j*, and *k* are commonly used to index vectors and matrices. Computer programmers make considerable use of `for` loops in array processing, so programmers have universally adopted this convention of short control variable names. Thus, it generally is acceptable to use simple identifiers like `i` as loop control variables.

The `break` and `continue` statements can be used in the body of a `for` statement. Like with the `while` and `do/while` statements, `break` causes immediate loop termination, and `continue` causes the condition to be immediately checked to determine if the iteration should continue. As previously mentioned, `for` loop control should be restricted to its header, and the use of `break` and `continue` should be avoided.

Any `for` loop can be rewritten with a `while` loop and behave identically. For example, consider the `for` loop

```
for ( int i = 1; i <= 10; i++ ) {
    System.out.println(i);
}
```

and next consider the `while` loop that behaves exactly the same way:

```
int i = 1;
while ( i <= 10 ) {
    System.out.println(i);
    i++;
}
```

Which is better? The `for` loop conveniently packages the loop control information in its header, but in the `while` loop this information is distributed throughout the small section of code. The `for` loop thus provides a better organization of the loop control code. Does one loop outperform the other? No. These two sections of code are compiled into *exactly* the same bytecode:

```
0 iconst_1          // ---+
1 istore_0          // ---+----> i = 1
2 goto 15           // -----> go to line 15
5 getstatic #2 <Field java.io.PrintStream out> // --+
8 iload_0           // -----+
9 invokevirtual #3 <Method void println(int)> // --+--> print i
12 iinc 0 1         // ---> i++
15 iload_0          // ---+
16 bipush 10        // ---+
18 if_icmple 5       // ---+----> if i <= 10 go to line 5
```

Thus, the `for` loop is preferred in this example.

19.5 Summary

- Add summary items here.

19.6 Exercises

Chapter 20

Arrays

Individual variables are classified as *scalars*. A scalar can assume exactly one value at a time. As we have seen, individual variables can be used to create some interesting and useful programs. Scalars, however, do have their limitations. Consider `AverageNumbers` (Figure 20.1) which averages five numbers entered by the user.

```
import java.util.Scanner;

public class AverageNumbers {
    public static void main(String[] args) {
        double n1, n2, n3, n4, n5;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter five numbers: ");
        // Allow the user to enter in the five values.
        n1 = scan.nextDouble();
        n2 = scan.nextDouble();
        n3 = scan.nextDouble();
        n4 = scan.nextDouble();
        n5 = scan.nextDouble();
        System.out.println("The average of " + n1 + ", " + n2
                           + ", " + n3 + ", " + n4 + ", " + n5
                           + " is " + (n1 + n2 + n3 + n4 + n5)/5);
    }
}
```

Listing 20.1: `AverageNumbers`—Average five numbers entered by the user

A sample run of `AverageNumbers` looks like:

```
Please enter five numbers: 9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

AverageNumbers (Figure 20.1) conveniently displays the values the user entered and then computes and displays their average. Suppose the number of values to average must increase from five to 25. Twenty additional variables must be introduced, and the overall length of the program will necessarily grow. Averaging 1,000 numbers using this approach is impractical.

AverageNumbers2 (Figure 20.2) provides an alternative approach for averaging numbers.

```
import java.util.Scanner;

public class AverageNumbers2 {
    private static int NUMBER_OF_ENTRIES = 5;
    public static void main(String[] args) {
        double sum = 0.0;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter " + NUMBER_OF_ENTRIES
            + " numbers: ");
        // Allow the user to enter in the five values.
        for (int i = 0; i < NUMBER_OF_ENTRIES; i++) {
            sum += scan.nextDouble();
        }
        System.out.println("The average of the " + NUMBER_OF_ENTRIES
            + " values is "
            + sum/NUMBER_OF_ENTRIES);
    }
}
```

Listing 20.2: AverageNumbers2—Another program that averages five numbers entered by the user

It behaves slightly differently from the AverageNumbers program, as the following sample run using the same data shows:

```
Please enter five numbers: 9 3.5 0.2 100 15.3
The average of the five values is 25.6
```

AverageNumbers2 can be modified to average 25 values much more easily than the AverageNumbers that must use 25 separate variables (just change the constant NUMBER_OF_ENTRIES). In fact, the coding change to average 1,000 numbers is no more difficult. However, unlike AverageNumbers, AverageNumbers2 does not display the numbers entered. This is a significant difference; it may be necessary to retain all the values entered for various reasons:

- All the values can be redisplayed after entry so the user can visually verify their correctness.
- The values may need to be displayed in some creative way; for example, they may be placed in a graphical user interface component, like a visual grid (spreadsheet).
- The values entered may need to be processed in a different way after they are all entered; for example, we may wish to display just the values entered above a certain value (like greater than zero), but the limit is not determined until after all the numbers are entered.

In all of these situations we must retain the values of all the variables for future recall.

We need to combine the advantages of both of the above programs:

- `AverageNumbers`—the ability to retain individual values
- `AverageNumbers2`—the ability to dispense with creating individual variables to store all the individual values

An *array* captures both of these advantages in one convenient package. An array is a *nonscalar* variable. An array is a collection of values. An array has a name, and the values it contains are accessed via their position within the array.

20.1 Declaring and Creating Arrays

An array is a nonscalar variable. Like any other variable, an array can be a local, class, or instance variable, and it must be declared before it is used. Arrays can be declared in various ways:

- The general form of a simple, single array declaration is

type [] *name*;

The square brackets denote that the variable is an array. Some examples of simple, single array declarations include:

```
int[] a;           // a is an array of integers
double[] list;     // list is an array of double-precision
                  // floating point numbers
```

- Multiple arrays of the same type can be declared as

type [] *name*₁, *name*₂, ..., *name*_n;

The following declaration declares three arrays of characters and two Boolean arrays:

```
char[] a, letters, cList;
boolean[] answerVector, selections;
```

In each of the above cases, the number of elements is not determined until the array is created. In Java, an array is a special kind of *object*. An array variable is therefore an *object reference*. Declaring an array does not allocate space for its contents just like declaring an object reference does not automatically create the object it is to reference; the `new` operator must be used to create the array with the proper size. Given the declarations above, the following examples show how arrays can be created:

```
// Declarations, from above
char[] a, letters, cList;
boolean[] answerVector, selections;
```

```

/* . . . */
// Array allocations
a = new char[10]; // a holds ten characters
letters = new char[scan.nextInt()]; // Number of characters
                                   // entered by user
cList = new char[100]; // cList holds 100 characters
answerVector = new Boolean[numValues]; // Size of answerVector
                                       // depends on the value
                                       // of a variable

```

An array has a declared type, and all elements stored in that array must be compatible with that type. Because of this, arrays are said to be *homogeneous* data structures.

Arrays may be initialized when they are declared; for example:

```
char[] a = new char[10];
```

This statement declares an array of characters and creates it with a capacity of ten elements. When an array is allocated, by default its contents are all *zero-like*. Zero-like means something different for different types, as Table 20.1 shows. Actually, these zero-like values are used as default values in two different situations:

Type	Value
byte, short, int, char, long, float, double	0
boolean	false
Object reference	null

Table 20.1: Zero-like values for various types

- they are assigned to elements of newly allocated arrays that are not otherwise initialized and
- they are assigned to class and instance variables that are not otherwise initialized.

It is possible to both create an array and initialize its contents simultaneously. The elements are provided in a comma separated list within curly braces:

$$type[] \textit{name} = \{ \textit{value}_1, \textit{value}_2, \dots, \textit{value}_3 \};$$

Examples include:

```

int[] a = { 20, 30, 40, 50 };
char[] cList = { 'a', 'b', 'c', 'd' };

```

This comma separated list of initial values is called an *initialization list*. No integer expression is used in the square brackets because the compiler can calculate the length of the initialization list. Notice that `new` is not used here even though an array object is being created on the heap; this special syntax is only possible when an array is declared with an initialization list. Variables can also be used with this special syntax:

```
int x = 10, y = 20, z = 30;
int[] a = { x, y, z };
```

Finally, it is possible to create an array with an initialization list sometime after it has been declared, as the following example shows:

```
int[] a;
/* . . . */
a = new int[] { 10, 20, 30 };
```

Notice that `new` operator must be used in this case.

Figure 20.1 illustrates how arrays are created.

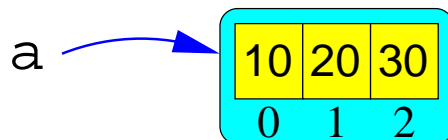
```
int[] a, b;
```

a → ?

b → ?

First, declare the array variables

```
a = new int[] { 10, 20, 30 };
b = new int[5];
```



Then create and assign the arrays

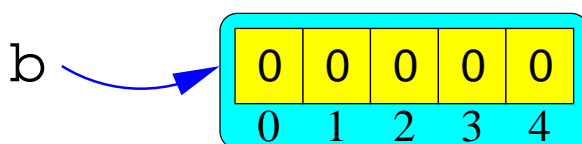


Figure 20.1: Creating arrays

20.2 Using Arrays

Once an array has been properly declared and created it can be used within a program. A programmer can use an array in one of two ways:

1. an element of the array can be used within a statement or

2. a reference to the whole array itself can be used within a statement.

In the next chapter we will see how an array as a whole can be used within a program; in this chapter we see how individual elements can be accessed.

Once an array has been created, its elements can be accessed with the `[]` operator. The general form of an array access expression is:

$$\textit{name} [\textit{expression}]$$

The expression within the square brackets must evaluate to an integer; some examples include

- an integer literal: `a[34]`
- an integer variable: `a[x]`
- an integer arithmetic expression: `a[x + 3]`
- an integer result of a method call that returns an `int`: `a[find(3)]`
- an access to an integer array: `a[b[3]]`

The square brackets and enclosed expression is called an *index* or *subscript*. The subscript terminology is borrowed from mathematicians who use subscripts to reference elements in a vector (for example, V_2 represents the second element in vector V). Unlike the convention used in mathematics, however, the first element in the array is at position zero, not one. The index indicates the distance from the beginning; thus, the very first element is at a distance of zero from the beginning of the array. The first element of array `a` is `a[0]`. If array `a` has been allocated to hold n elements, then the last element in `a` is `a[n - 1]`. Figure 20.2 illustrates assigning an element in an array. The square bracket

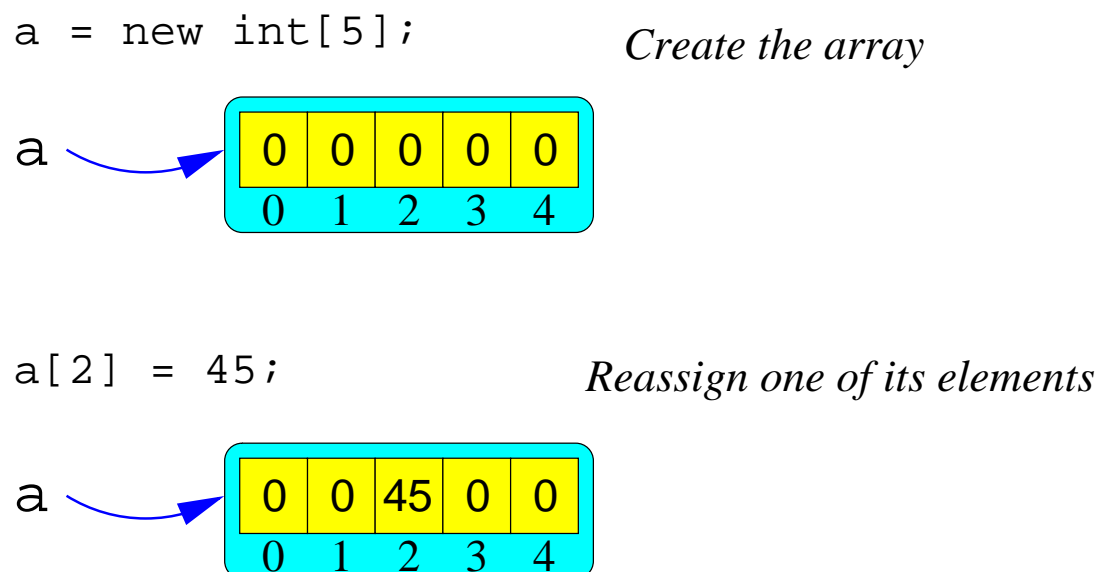


Figure 20.2: Assigning an element in an array

notation is one aspect that makes arrays special objects in Java. An object of any other class must use a method call

to achieve the same effect; for example, the `charAt()` method must be used to access individual characters within a `String` object.

The programmer must ensure that the index is within the bounds of the array. Since the index can consist of an arbitrary integer expression whose value cannot be determined until run time, the compiler cannot check for out-of-bound array accesses. A runtime error will occur if a program attempts an out-of-bounds array access. The following code fragments illustrate proper and improper array accesses:

```
double[] numbers = new double[10]; // Declare and allocate array
numbers[0] = 5; // Put value 5 first
numbers[9] = 2.5; // Put value 2.5 last
numbers[-1] = 5; // Runtime error; any negative index is illegal
numbers[10] = 5; // Runtime error; last valid position is 9
numbers[1.3] = 5; // Compile-time error; nonintegral index illegal
```

Notice that the compiler does check that the type of the index is correct.

Loops are frequently used to access the values in an array. `ArrayAverage` (Listing 20.3) uses an array and a loop to achieve the generality of `AverageNumbers2` and the ability to retain all input for later redisplay:

```
import java.util.Scanner;

public class ArrayAverage {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double[] numbers = new double[5];
        double sum = 0.0;
        System.out.print("Please enter five numbers: ");
        // Allow the user to enter in the five values.
        for (int i = 0; i < 5; i++) {
            numbers[i] = scan.nextDouble();
            sum += numbers[i];
        }
        System.out.print("The average of ");
        for (int i = 0; i < 4; i++) {
            System.out.print(numbers[i] + ", ");
        }
        // No comma following last element
        System.out.println(numbers[4] + " is " + sum/5);
    }
}
```

Listing 20.3: `ArrayAverage`—Use an array to average five numbers entered by the user

The output of `ArrayAverage` is identical to our original `AverageNumbers` (Listing 20.1) program:

```
Please enter five numbers:  9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

but we can conveniently extend it to handle as many values as we wish.

Notice in `ArrayAverage` that the first loop collects all five input values from the user. The second loop only prints the first four because it also prints a trailing comma after each element; since no comma is to follow the last element, it is printed outside the loop.

`ArrayAverage` is less than perfect, however. In order to modify it to be able to handle 25 or 1,000 elements, no less than five lines of source code must be touched (six, if the comment is to agree with the code!). A better version is shown in `BetterArrayAverage` (Figure 20.4):

```
import java.util.Scanner;

public class BetterArrayAverage {
    private static final int SIZE = 5;
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double[] numbers = new double[SIZE];
        double sum = 0.0;
        System.out.print("Please enter " + SIZE + " numbers: ");
        // Allow the user to enter in the values.
        for (int i = 0; i < SIZE; i++) {
            numbers[i] = scan.nextDouble();
            sum += numbers[i];
        }
        System.out.print("The average of ");
        for (int i = 0; i < SIZE - 1; i++) {
            System.out.print(numbers[i] + ", ");
        }
        // No comma following last element
        System.out.println(numbers[SIZE - 1] + " is " + sum/SIZE);
    }
}
```

Listing 20.4: `BetterArrayAverage`—An improved `ArrayAverage`

`BetterArrayAverage`'s output is only slightly different from that of `ArrayAverage`:

```
Please enter 5 numbers:  9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

In `BetterArrayAverage`, only the definition of the constant `SIZE` must be changed to allow the program to average a different number of values. This centralization of the definition of the array's size eliminates redundancy and leads to a program that is more maintainable. When redundant information is scattered throughout a program, it is a common error to update some but not all of the information when a change is to be made. If all of the redundant information is not updated to agree, the inconsistencies result in errors within the program.

Sometimes it is necessary to determine the size of an array. All arrays have a public final attribute named `length` that holds the allocated size of the array. For example, the first for loop in `BetterArrayAverage` could be rewritten as

```
for ( int i = 0; i < numbers.length; i++ ) {
    numbers[i] = scan.nextDouble();
    sum += numbers[i];
}
```

Note that the last element in `numbers` is `numbers[numbers.length - 1]`. This feature is valuable when an array must be manipulated by code that did not allocate it (see § 21.1 for such an example).

A common idiom for visiting each element in an array `a` is

```
for ( int i = 0; i < a.length; i++ ) {
    // do something with a[i]
}
```

If the elements of `a` are only being read and not modified, a variant `for` construct called the “for/each” loop can be used. If, for example, array `a` held doubles, the above `for` loop could be rewritten

```
for ( double d : a ) {
    // do something with d
}
```

You read the above statement as “for each double `d` in `a`, do something with `d`.” Inside the body of the loop `d` represents a particular element of `a`: the first time through `d = a[0]`, the second time through `d = a[1]`, etc. The last time through the loop `d` represents `a[a.length - 1]`, the last element in `a`.

Where possible, the for/each construct should be used instead of the normal `for` construct. The for/each construct is simpler and has fewer pieces for the programmer to get wrong. The for/each loop cannot be used all the time, though:

- Since `d` is variable local to the for/each loop and `d` is a copy of an element in `a`, modifying `d` does not affect the contents of `a`. The statement

```
for ( int i = 0; i < a.length; i++ ) {
    System.out.println(a[i]);
}
```

can be readily transformed to the equivalent but simpler

```
for ( double d : a ) {
    System.out.println(d);
}
```

but the statement

```
for ( int i = 0; i < a.length; i++ ) {
    a[i] = 0;
}
```

cannot be transformed into

```

    for ( double d : a ) {
        d = 0;
    }

```

since the `for/each` version sets the copy of the array element to zero, not the array element itself.

- The `for/each` loop visits every element in the array. If only a subrange of the array is to be considered, the `for/each` loop is not appropriate and the standard `for` loop should be used.
- The `for/each` loop visits each element in order from front to back (lowest index to highest index). If array elements must be considered in reverse order or if not all elements are to be considered, the `for/each` loop cannot be used.
- The `for/each` statement does not reveal in its body the index of the element it is considering. If an algorithm must know the index of a particular element to accomplish its task, the traditional `for` loop should be used. The traditional `for` loop's control variable also represents inside the body of the loop the index of the current element.

20.3 Arrays of Objects

An array can store object references. Recall the simple class `Widget` (Figure 9.2) from Section 9.4. The following code

```
Widget[] widgetList = new Widget[10];
```

declares and creates an array of ten `Widget` references. It does not create the individual objects in the array. In fact, the elements of `widgetList` are all null references. The following code

```

Widget[] widgetList = new Widget[10];
for ( int w : widgetList ) {
    w.identify();
}

```

will compile fine but fail during execution with a *Null pointer exception*, since all the elements are null. It is an error to attempt to dereference a null pointer (in this case `null.identify()`). If `widgetList` is indeed to hold ten `Widget` objects, these objects must be created individually. Since the `Widget` constructor takes no arguments, this can easily be done in a loop:

```

Widget[] widgetList = new Widget[10];
for ( int i = 0; i < widgetList.length; i++ ) {
    widgetList[i] = new Widget();
}

```

This code ensures that `widgetList` holds ten viable `Widget` objects.

Because `Object` is the root of the Java class hierarchy, any reference type *is a* `Object`. This means any reference type (`String`, `TrafficLightModel`, `Random`, etc.) is assignment compatible with `Object`, as illustrated by the following legal code fragment:

```
Object obj = "Hello"; // OK, since a string IS an object
```


An array of `Object`s, therefore, can hold references of *any* type:

```
Object[] obj = { "Hello", new Random(),
                new TrafficLightModel(TrafficLightModel.STOP),
                new Rational(1, 2) };
```

Furthermore, since all primitive types have associated wrapper classes, and since primitive values are autoboxed when required (see § 13.4), the following code is legal as well:

```
Object[] obj = { "Hello", new Random(), 12,
                new Rational(1, 2), 3.14, "End" };
```

We said that arrays hold *homogeneous* types; for example, an array of `int`s can only hold `int` values. This is true, but because anything is assignment compatible with `Object`, an `Object` array can hold a diverse collection of *heterogenous* values. Of course the values are heterogenous with respect to each other, but they are homogeneous in the fact that they are all `Object`s. Their behavior within the array is differentiated by polymorphism:

```
// obj is the array of Objects from above
for (Object elem : obj) {
    System.out.println(elem);
}
```

The `toString()` method in each element determines how each element is displayed.

Since an array is itself an object reference, arrays of arrays are possible. Section 20.4 explores arrays of arrays, commonly called multidimensional arrays.

20.4 Multidimensional Arrays

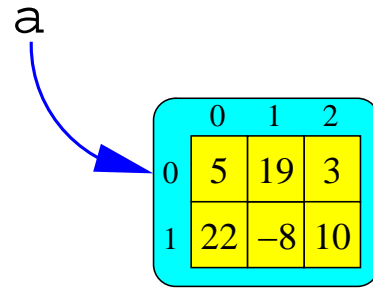
Java supports arrays containing any type—even arrays of arrays. The statement

```
int[][] a;
```

declares `a` to be an array of integer arrays. It is convenient to visualize this array of arrays as a two-dimensional (2D) structure like a table. A 2D (or higher dimension) array is also called a *matrix*. Figure 20.3 shows a picture of the array created by the following sequence of code:

```
int[][] a;
a = new int[2][3]; // Creates a 2D array filled with zeros
a[0][0] = 5;
a[0][1] = 19;
a[0][2] = 3;
a[1][0] = 22;
a[1][1] = -8;
a[1][2] = 10;
```

The two-dimensional array `a` is said to be a 2×3 array, meaning it has two rows and three columns (as shown in Figure 20.3). The first index signifies the row and the second index denotes the column of the element within the array.

Figure 20.3: A 2×3 two-dimensional array

Using a syntax similar to the initialization lists of one-dimensional arrays, `a` above could have been declared and created as:

```
int[][] a;
a = new int[][] { new int[] { 5, 19, 3 },
                  new int[] { 22, -8, 10 } };
```

or even more compactly declared and initialized as:

```
int[][] a = { { 5, 19, 3 },
              { 22, -8, 10 } };
```

Each element of a 2D array is a 1D array. Thus, if `a` is a 2D array and `i` is an integer, then the expression

```
a[i]
```

is a 1D array. To obtain the scalar element at position `j` of this array `a[i]`, double indexing is required:

```
a[i][j]    // The element at row i, column j
```

The following statement assigns the value 10 to the element at row 1, column 2:

```
a[1][2] = 10;
```

Since 2D arrays in Java are really arrays of arrays and are not tables, rows are not required to contain the same number of elements. For example, the following statement

```
int[][] a = { { 5, 19, 3 },
              { 22, -8, 10, 14, 0, -2, 8 },
              { 20, -8 },
              { 14, 15, 0, 4, 4 } };
```

creates what is known as a *jagged table* with unequal row sizes. Figure 20.4 illustrates this jagged table.

Since a 2D array is an array of 1D arrays and each of these 1D arrays are treated as rows, the `length` attribute of a 2D array specifies the number of rows. Each element of the 2D array represents a row which is itself an array with its own `length` attribute. Thus, given the declaration:

	0	1	2	3	4	5	6
0	5	19	3				
1	22	-8	10	14	0	-2	8
2	20	-8					
3	14	15	0	4	4		

Figure 20.4: A “jagged table” where rows have unequal lengths

```
int[][] a;
```

- `a.length` represents the number of rows in array `a`
- `a[i]` represents row `i`, itself an array
- `a[i].length` is the number of elements in row `i`
- `a[i][j]` represents an integer element at row `i`, column `j`

The following method displays the contents of a 2D integer array:

```
public static void print2Darray(int[][] a) {
    for ( int row = 0; row < a.length; row++ ) {
        for ( int col = 0; col < a[row].length; col++ ) {
            System.out.print(a[row][col] + " ");
        }
        System.out.println(); // Newline for next row
    }
}
```

The outer loop iterates over the rows (`a.length` is the number of rows); the inner loop iterates over the elements (columns) within each row (`a[row].length` is the number of elements in row `row`).

Arrays with dimensions higher than two can be represented. A 3D array is simply an array of 2D arrays, a 4D array is an array of 3D arrays, etc. For example, the statement

```
matrix[x][y][z][t] = 1.0034;
```

assigns 1.0034 to an element in a 4D array of doubles. In practice, arrays with more than two dimensions are rare, but advanced scientific and engineering applications sometimes require higher-dimensional arrays.

20.5 Summary

- Add summary items here.

20.6 Exercises

Chapter 21

Arrays as Objects

An array is simply an object reference (see Section 20.1). As such, it can be used like any object reference; an array can be

- passed as an actual parameter to a method and
- returned from a method as a return value.

In the previous chapter we saw how access and work with the elements within an array. In this chapter we investigate how manipulate an array itself as a whole.

21.1 Array Parameters

Array parameters. An array is specified as a formal parameter in a method definition with the same syntax as a variable declaration:

```
// Method f accepts two arrays and a Boolean
public int f(int[] numList, double[] vector, flag) {
    /* . . . Details omitted . . . */
}
```

Calling code simply passes the array as it would any variable:

```
int[] list = new int[100];
double[] v = new double[3];
/* Initialize list and v (details omitted) */
// . . . then call f()
list[3] = f(list, v, true);
```

The following method sums the elements of an integer array:

```
public static int sumUp(int[] a) {
    int sum = 0;
```

```

        for ( int value : a ) {
            sum += value;
        }
        return sum;
    }

```

Since the method accepts any integer array with no additional information about its size, the number of elements in the array must be determined from the array's `length` attribute. The `for/each` statement implicitly examines the `length` field to iterate the correct number of times.

The parameter passing mechanism for array variables works just like it does for other reference variables. Suppose the `sumUp()` method is called passing an integer array named `list`:

```

int[] list;
/* Allocate and initialize list (details omitted) */
// Display the sum of all the elements in list
System.out.println(sumUp(list));

```

While in the `sumUp()` method, `a` is an alias of `list`. This means that modifying `a` within `sumUp()` also affects `list` in the calling environment.

It is not necessary to use a named array as an actual parameter. The following example makes up an *anonymous array* and passes it to the `sumUp()` method:

```

// Display the sum of all the elements the anonymous array specified here
System.out.println(sumUp(new int[] { 10, 20, 30, 40, 50 }));

```

21.2 Copying an Array

It is important to note that an array is simply an object reference. It may seem plausible to make a copy of an array as follows:

```

int[] a, b;    // Declare two arrays
a = new int[] { 10, 20 , 30 }; // Create one
b = a;         // Make a copy of array a?

```

Since an array is an object reference, and since `b` has been assigned to `a`, `a` and `b` refer to the same array. Array `b` is an alias of `a`, not a copy of `a`. Figure 21.1 illustrates this array aliasing.

The following code can be used to make a copy of `a`:

```

int[] a, b;    // Declare two arrays
a = new int[] { 10, 20 , 30 }; // Create one
// Really make a copy of array a
b = new int[a.length]; // Allocate b
for ( int i = 0; i < a.length; i++ ) {
    b[i] = a[i];
}

```

A new array must be created, and then each element is copied over into the new array, as shown in Figure 21.2.

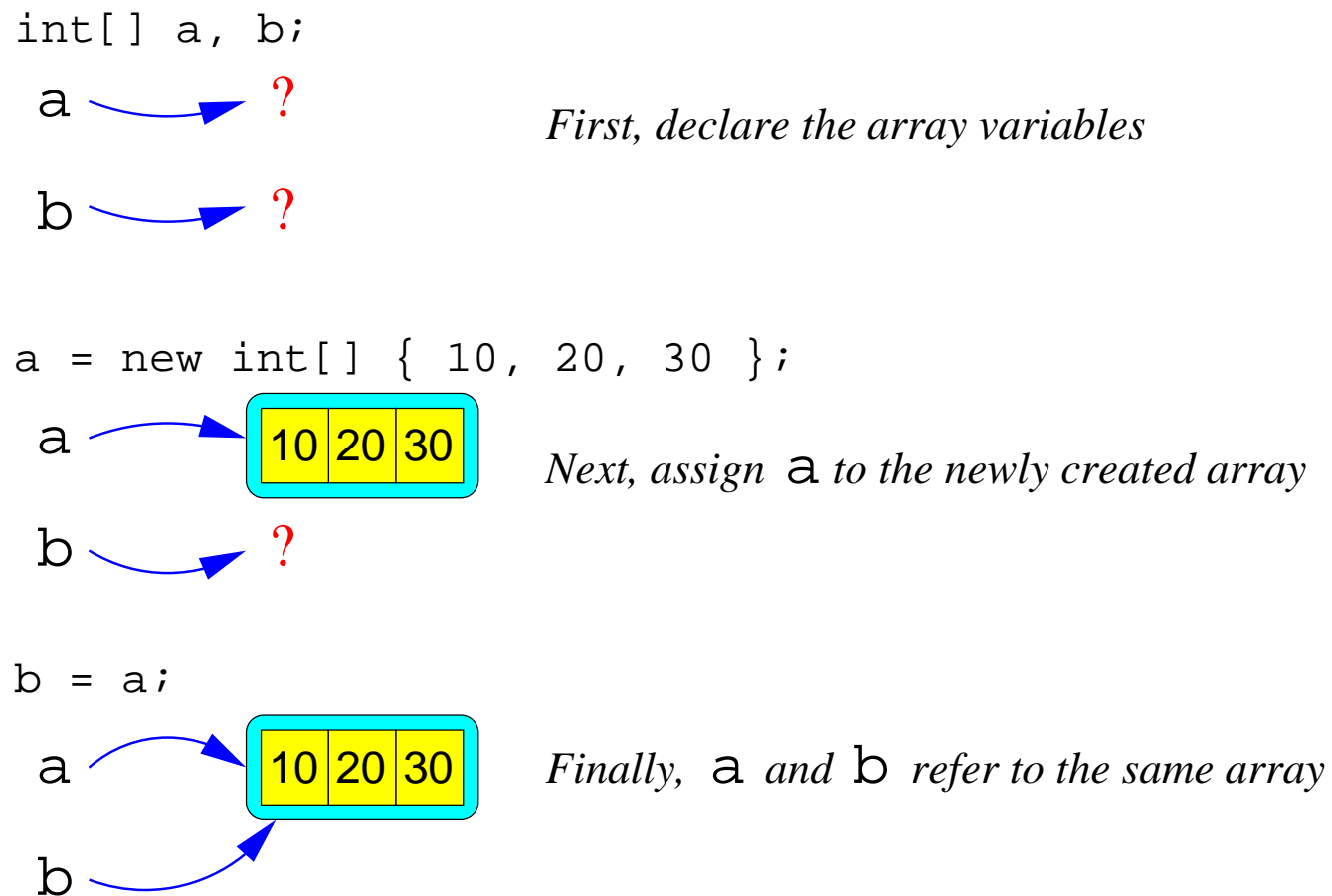


Figure 21.1: Array aliasing

Since copying an array is such a common activity, the `java.lang.System` class has a method named `arraycopy()` that makes it convenient to copy all or part of the elements of one array to another array. It requires five parameters:

1. The source array; that is, the array *from* which the elements are to be copied
2. The starting index in the source array from which the elements will be copied
3. The destination array; that is, the array *into* which the elements are to be copied
4. The starting index in the destination array into which the elements will be copied
5. The number of elements to be copied

Notice that this method is quite flexible. All or part of an array may be copied into another array. The simplest case is a complete copy, as in the example above:

```
int[] a, b;    // Declare two arrays
a = new int[] { 10, 20, 30 }; // Create one
// Make a copy of array a
b = new int[a.length]; // Allocate b
System.arraycopy(a, 0, b, 0, b.length);
```

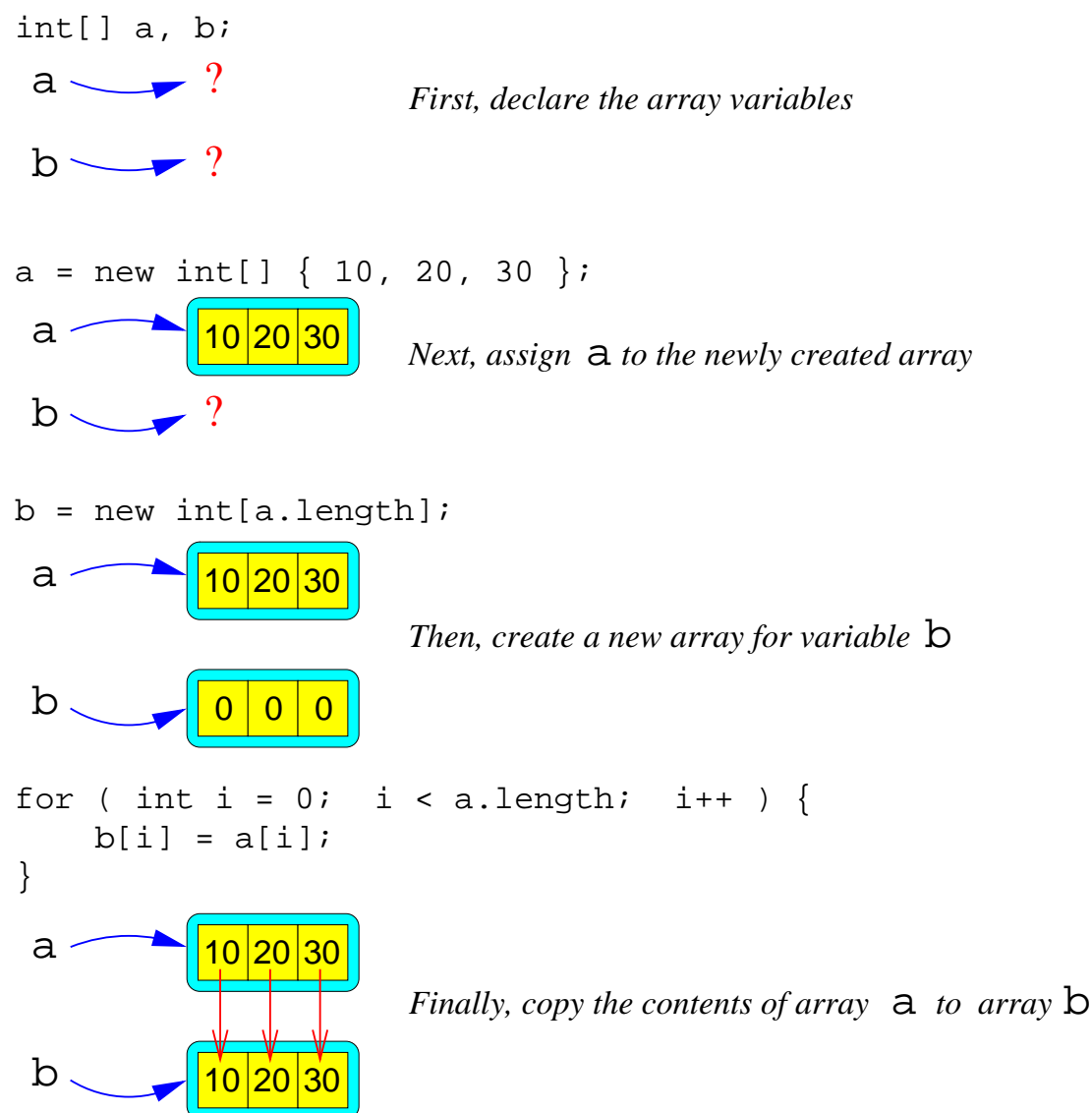


Figure 21.2: Making a true copy of an array

In the following case part of one array is copied into part of another array:

```
int[] a = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
int[] b = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
System.arraycopy(b, 2, a, 4, 3);
```

After executing this code the contents of *a* are:

10,20,30,40,33,44,55,80,90

The `System.arraycopy()` method is implemented in such a way to be more efficient than equivalent code written in the Java language. Table 21.1 shows the results of five runs of `ArrayCopyBenchmark` (Table 21.1) on an array of 5,000,000 elements. On average, `System.arraycopy()` is only 2% faster than the pure Java copy code.

```
public class ArrayCopyBenchmark {
    public static void main(String[] args) {
```


Run Number	for Loop (msec)	arraycopy (msec)	Percent Speedup
1	137	130	
2	137	134	
3	115	112	
4	137	145	
5	137	129	
Average	132.6	130.0	2.0

Table 21.1: Empirical data comparing a straightforward array copy to `System.arraycopy()`. The array size is 5,000,000.

```

if ( args.length < 1 ) {
    System.out.println("Usage:");
    System.out.println("    java ArrayCopyBenchmark <size>");
    System.exit(1);
}
int size = Integer.parseInt(args[0]);
// Initialize the source array with random values
int[] src = new int[size];
java.util.Random rand = new java.util.Random();
for ( int i = 0; i < size; i++ ) {
    src[i] = rand.nextInt(size);
}
int[] dest = new int[size];
Stopwatch timer = new Stopwatch();
// Copy array by hand
timer.start();
for ( int i = 0; i < size; i++ ) {
    dest[i] = src[i];
}
timer.stop();
System.out.println("Elapsed time: " + timer.elapsed());
// Copy array using System.arraycopy()
timer.start();
System.arraycopy(src, 0, dest, 0, size);
timer.stop();
System.out.println("Elapsed time: " + timer.elapsed());
}

```

Listing 21.1: `ArrayCopyBenchmark`—compares a straightforward array copy to `System.arraycopy()`

21.3 Array Return Values

Array return values. An array can be returned by a method, as can any reference type. `PrimesList` (Figure 21.2) returns an array of prime numbers over a given range.

```
import java.util.Scanner;

public class PrimesList {
    public static boolean isPrime(int n) {

        boolean result = true; // Assume no factors unless we find one
        for ( int trialFactor = 2; trialFactor <= Math.sqrt(n); trialFactor++ ) {
            if ( n % trialFactor == 0 ) { // Is trialFactor a factor?
                result = false;
                break; // No need to continue, we found a factor
            }
        }
        return result;
    }

    // Returns a list (array) of primes in the range start...stop
    public static int[] generatePrimes(int start, int stop) {
        // First, count how many there are
        int value = start; // Smallest potential prime number
        int count = 0; // Number of primes in the list
        while ( value <= stop ) {
            // See if value is prime
            if ( isPrime(value) ) {
                count++;
            }
            value++; // Try the next potential prime number
        }
        // Next, create an array exactly the right size
        int[] result = new int[count];
        // Next, populate the array with the primes
        count = 0;
        value = start; // Smallest potential prime number
        while ( value <= stop ) {
            // See if value is prime
            if ( isPrime(value) ) {
                result[count++] = value;
            }
            value++; // Try the next potential prime number
        }
        return result;
    }

    public static void main(String[] args) {
        int maxValue;
        Scanner scan = new Scanner(System.in);
        System.out.print("Display primes up to what value? ");
        int[] primes = generatePrimes(2, scan.nextInt());
    }
}
```

```
    for ( int p : primes ) {  
        System.out.print (p + " ");  
    }  
    System.out.println();    // Move cursor down to next line  
}  
}
```

Listing 21.2: PrimesList—Uses a method to generate a list (array) of prime numbers over a given range

The `generatePrimes()` method first counts the number of prime numbers so that an array of the proper size can be allocated. Then it repeats the process again filling in each position of the newly allocated array with a prime number. Note that the statement

```
result[count++] = value;
```

uses the postincrement operator; thus, the current value of `count` is used as the index, then `count` is incremented.

21.4 Command Line Arguments

The `main()` method for an executable class has the general structure:

```
public static void main(String[] args) {  
    /* Body goes here . . . */  
}
```

The `main()` method specifies a single parameter—an array of strings. The identifier `args` may be replaced with any valid identifier name, but all the other names (`public`, `static`, `void`, `main`, and `String`) must appear as presented here. To this point we have ignored this parameter of `main()`, but now, armed with the knowledge of arrays, we can write even more flexible Java programs.

The *command line* is the sequence of text a user enters in a *command shell*. On a Unix/Linux system, the shell (typically `bash` or `csh`) is the command shell; on a Windows system, the DOS shell is a command shell. The command to see the files and subdirectories (subfolders) in the current directory (folder) is

```
ls
```

on a Unix/Linux system, and

```
dir
```

on a Windows system. Both `ls` and `dir` are considered simple command lines. Additional information can be provided to produce more complicated command lines. For example,

```
ls -l
```

on a Unix/Linux system gives a “long” listing providing extra information. The command line

```
dir /w
```

on a Windows system provides a “wide” listing of files. We say that `-l` is an argument to `ls` and `/w` is an argument to `dir`.

To execute the Java program `ArrayAverage` (Figure 20.3) from the command shell, the file `ArrayAverage.java` would first be compiled to produce the class file `ArrayAverage.class`. The Java interpreter (normally named `java`) would then be invoked as

```
java ArrayAverage
```

This is how a Java application is typically executed from the command line.

It is possible to pass additional information to Java programs using command line arguments. The command shell (OS) uses the string array parameter in the `main()` to pass command line arguments to a Java program. `CmdLineAverage` (Figure 21.3) accepts the values to average from the command line instead of having the user enter them during the program execution.

```
public class CmdLineAverage {
    public static void main(String[] args) {
        double sum = 0.0;
        // Get command line arguments and compute their average
        for ( String s : args ) {
            sum += Double.parseDouble(s);
        }
        System.out.print("The average of ");
        for ( int i = 0; i < args.length - 1; i++ ) {
            System.out.print(args[i] + ", ");
        }
        // No comma following last element
        System.out.println(args[args.length - 1] + " is "
                           + sum/args.length);
    }
}
```

Listing 21.3: `CmdLineAverage`—Averages values provided on the command line

Issuing the command line

```
java CmdLineAverage 9 3.5 0.2 100 15.3
```

yields the output

The average of 9, 3.5, 0.2, 100, 15.3 is 25.6

Figure 21.3 shows the correspondence of command line arguments to the `args` array elements.

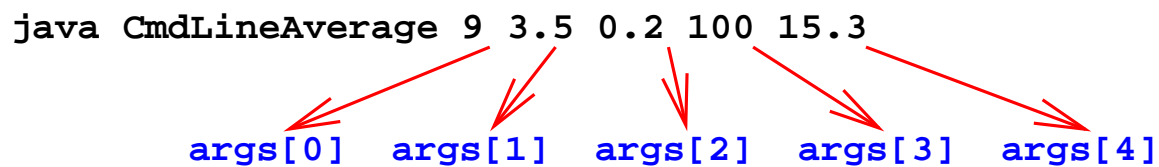


Figure 21.3: Correspondence of command line arguments to `args` array elements

It is important to note that command line arguments are whitespace delimited and each argument is presented to the JVM as a `String` object. As shown in `CmdLineAverage`, if a number is provided on the command line and the argument is to be processed as a number by the Java program, then the argument must be converted from its string form to its numeric form. In `CmdLineAverage`, the `Double.parseDouble()` class method is used to convert a string argument to a double.

At last, all aspects of the declaration of the `main()` method have been revealed:

- `public`—the `main()` method can be invoked by methods outside the class
- `static`—no object need be created to execute the `main()` method
- `void`—`main()` returns no value
- `main`—the name of the method invoked when the JVM executes the class
- `String[] args`—`main()` accepts an array of `String` objects as a parameter; these strings correspond to command line arguments

In IDEs developers typically run a Java program through a menu item or hotkey. In these development environments, command line arguments are often specified in a dialog box.

21.5 Variable Argument Lists

It is easy to write a method that adds two integers together and returns their sum. So too, three integers, four integers, etc. up to some small practical limit. Since Java permits methods to be overloaded, the task is easy, although time consuming. `OverloadedSumOfInts` (Figure 21.4) provides a good starting point.

```
public class OverloadedSumOfInts {
    public static int sum(int a, int b) {
        return a + b;
    }
    public static int sum(int a, int b, int c) {
        return a + b + c;
    }
}
```

```

public static int sum(int a, int b, int c, int d) {
    return a + b + c + d;
}
public static int sum(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}
// . . . Overload as many times as you like!
}

```

Listing 21.4: OverloadedSumOfInts—A simple class to sum integers

What if the number of integers to add exceeds the number of parameters in any of our overloaded methods? We could always use functional composition, as in

```

int total = OverloadedSumOfInts.sum(OverloadedSumOfInts.sum(3, 4, -1, 18, 6),
                                   OverloadedSumOfInts.sum(2, 2, 10, -3));

```

but this cumbersome. We could use an array and write just one method, as is ArraySumOfInts (Listing 21.5).

```

public class ArraySumOfInts {
    public static int sum(int[] a) {
        int sum = 0;
        for (int i : a) {
            sum += i;
        }
        return sum;
    }
}

```

Listing 21.5: ArraySumOfInts—A “better” class to sum integers using an array

The array version is extremely flexible, but it is more awkward for the client to use. Consider the code required to sum five selected integers:

```

int total = ArraySumOfInts.sum(new int[] { 3, 4, -1, 18, 6 });

```

The explicit array creation is necessary, but it sure would be convenient to just pass the integer arguments by themselves without the array wrapper.

Before Java 5.0, the number of arguments accepted by a method was fixed at compile time. A method declared with three formal parameters accepted only three actual parameters. As we have seen, overloaded methods provide additional flexibility, but even then the number of parameters is fixed. Java 5.0 provides a feature called *varargs* (the name can be traced to the C programming language) that allows method writers to leave the number of parameters open. The syntax is illustrated in VarargsSumOfInts (Listing 21.6).

```
public class VarargsSumOfInts {
    public static int sum(int... args) {
        int sum = 0;
        for (int i : args) {
            sum += i;
        }
        return sum;
    }
}
```

Listing 21.6: VarargsSumOfInts—A truly better class to sum integers using varargs

```
public class VarargTester {
    public static void main(String[] args) {
        System.out.println(OverloadedSumOfInts.sum(3, 4, 5, 6));
        System.out.println(ArraySumOfInts.sum(new int[] { 3, 4, 5, 6 }));
        System.out.println(VarargsSumOfInts.sum(3, 4, 5, 6));
    }
}
```

Listing 21.7: VarargTester—A truly better class to sum integers using varargs

Note that the body of `VarargsSumOfInts` is functionally identical to `ArraySumOfInts`; only the method signatures differ. The two methods work the same way—they process an array of `ints`. The new parameter specification has the following form:

typename . . . parametername

where *typename* is any standard or programmer-defined type (primitive or reference), and *parametername* is a programmer chosen parameter name. The `. . .` signifies that within the method body the parameter is to be treated as an array of the type specified. In our `VarargsSumOfInts` example, the parameter `args` is treated as an `int` array. The `. . .` also means that the client will send any number of parameters of the specified type. The compiler will generate code in the context of the caller that creates an array to hold these arguments and sends that new array off to the method to be processed. The client's view is now much simpler:

```
int total = VarargsSumOfInts.sum(3, 4, -1, 18, 6);
```

The call to `VarargsSumOfInts.sum()` now looks like a call to a normal method that accepts five parameters. Behind the scenes, however, an array is involved, and only one true parameter is passed.

The `VarargsSumOfInts.sum()` method accepts any number of `int` arguments, but what if you need to accept any number of arguments of mixed types? The solution is to expect any number of `Objects`, as in

```
public static void process(Object... objs) {
    /* Do something with parameters. . . */
}
```

Primitive types passed by the caller will be autoboxed into the appropriate wrapper objects. Of course, if nonuniform types are passed, the method body will be more tricky to write, since any type could appear at any place in the parameter list. As an example, the `System.out.printf()` method can accept any number of any types of parameters following its format string. Its signature is

```
public PrintStream printf(String format, Object... args)
```

The code within the body of `printf()` scans the control codes within the format string to properly access the array of Objects that follow. That is why `printf()` can fail with a runtime error if the programmer is inconsistent with placement of control codes and arrangement of trailing parameters.

As shown in `System.out.printf()`, normal parameters can be mixed with varargs. In order for the compiler to make sense of the call of such a method, all normal parameters must precede the single varargs parameter.

21.6 Summary

- Add summary items here.

21.7 Exercises

Chapter 22

Working with Arrays

This chapter introduces fundamental algorithms for arrays—sorting and searching—and then finishes with a sample application that uses arrays.

22.1 Sorting Arrays

Array sorting—arranging the elements within an array into a particular order—is a common activity. For example, an array of integers may be arranged in ascending order (that is, from smallest to largest). An array of words (strings) may be arranged in lexicographical (commonly called alphabetic) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to implement.

The selection sort algorithm is relatively simple and efficient. It works as follows:

1. Let A be the array and let $i = 0$.
2. Examine each element that follows position i in the array. If any of these elements is less than $A[i]$, then exchange $A[i]$ with the smallest of these elements. This ensures that all elements after position i are greater than or equal to $A[i]$.
3. If $i < A.length - 1$, then set i equal to $i + 1$ and goto Step 2.

If the condition in Step 3 is not met, the algorithm terminates with a sorted array. The command to “goto Step 2” in Step 3 represents a loop. The directive at Step 2 to “Examine each element that follows ...” must also be implemented as a loop. Thus, a pair of nested loops is used in the selection sort algorithm. The outer loop moves a pointer from position zero to the next-to-the-last position in the array. During each iteration of this outer loop, the inner loop examines each element that follows the current position of the outer loop’s pointer.

SortIntegers (Figure 22.1) sorts the list of integers entered by the user on the command line.

```
public class SortIntegers {
    public static void selectionSort(int[] a) {
        for ( int i = 0; i < a.length - 1; i++ ) {
            int small = i;
            // See if a smaller value can be found later in the array
```

```

        for ( int j = i + 1; j < a.length; j++ ) {
            if ( a[j] < a[small] ) {
                small = j; // Found a smaller value
            }
        }
        // Swap a[i] and a[small], if a smaller value was found
        if ( i != small ) {
            int tmp = a[i];
            a[i] = a[small];
            a[small] = tmp;
        }
    }
}

public static void print(int[] a) {
    if ( a != null ) {
        for ( int i = 0; i < a.length - 1; i++ ) {
            System.out.print(a[i] + ", ");
        }
        // No comma following last element
        System.out.println(a[a.length - 1]);
    }
}

public static void main(String[] args) {
    // Convert the arrays of strings into an array of integers
    int[] numberList = new int[args.length];
    for ( int i = 0; i < args.length; i++ ) {
        numberList[i] = Integer.parseInt(args[i]);
    }
    selectionSort(numberList); // Sort the array
    print(numberList);         // Print the results
}
}

```

Listing 22.1: SortIntegers—Sorts the list of integers entered by the user on the command line

The command line

```
java SortIntegers 4 10 -2 8 11 4 22 9 -5 11 45 0 18 60 3
```

yields the output

```
-5, -2, 0, 3, 4, 4, 8, 9, 10, 11, 11, 18, 22, 45, 60
```

SortIntegers uses its `selectionSort()` method to physically rearrange the elements in the array. Since this `selectionSort()` method is a public class method (it is `static` and therefore cannot access any instance variables), it can be used by methods in other classes that need to sort an array of integers into ascending order. Selection sort

is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort. One such general purpose sort is *Quicksort*, devised by C. A. R. Hoare in 1962. Quicksort is the fastest sort for most applications. Since sorting is a common activity, Java provides a standard library class, `java.util.Arrays`, that implements Quicksort. Among other useful methods, the `Arrays` class contains a number of overloaded `sort()` class methods that can sort many different types of arrays. To use this standard Quicksort in the `SortIntegers` program (Figure 22.1), simply replace the statement

```
selectionSort(numberList); // Sort the array
```

with

```
java.util.Arrays.sort(numberList); // Sort the array
```

or, if `java.util.Arrays` is imported, just

```
Arrays.sort(numberList); // Sort the array
```

The results will be the same, but for large arrays, the Quicksort version will be much faster. `CompareSorts` (Figure 22.2) compares the execution times of our selection sort to Quicksort.

```
import java.util.Random;
import java.util.Arrays;

public class CompareSorts {
    public static void selectionSort(int[] a) {
        for ( int i = 0; i < a.length - 1; i++ ) {
            int small = i;
            // See if a smaller value can be found later in the array
            for ( int j = i + 1; j < a.length; j++ ) {
                if ( a[j] < a[small] ) {
                    small = j; // Found a smaller value
                }
            }
            // Swap a[i] and a[small], if a smaller value was found
            if ( i != small ) {
                int tmp = a[i];
                a[i] = a[small];
                a[small] = tmp;
            }
        }
    }

    public static void main(String[] args) {
        // Program must have size to be able to run
        if ( args.length != 1 ) {
            System.out.println("Usage:");
            System.out.println("    java CompareSorts <array_size>");
            System.exit(1);
        }
        // Allocate the arrays from size provided on the command line
        int[] listSS = new int[Integer.parseInt(args[0])];
```

```

int[] listQS = new int[listSS.length];
// Create a random number generator
Random random = new Random();
// Initialize the arrays with identical random integers
for ( int i = 0; i < listSS.length; i++ ) {
    int randomValue = random.nextInt();
    listSS[i] = listQS[i] = randomValue;
}
long timeSS = 0, timeQS = 0;
Stopwatch timer = new Stopwatch();
// Time selection sort
timer.start();
selectionSort(listSS);
timer.stop();
timeSS = timer.elapsed();
timer.start();
// Time Java's Quicksort
Arrays.sort(listQS);
timer.stop();
timeQS = timer.elapsed();
// Report results
System.out.println("Array size = " + args[0]
                  + "    Selection sort: " + timeSS
                  + "    Quicksort: " + timeQS);
}
}

```

Listing 22.2: CompareSorts—Compares the execution times of selection sort versus Quicksort

Table 22.1 shows the results of running CompareSorts.

Our selection sort is faster than Quicksort for arrays of size 500 or less. Quicksort performs much better for larger arrays. A 100,000 element array requires about four minutes to sort with our selection sort, but Quicksort can sort the same array in about one-fifth of a second! Section 22.2 addresses performance issues in more detail.

22.2 Searching Arrays

Searching an array for a particular element is a common activity. LinearSearch (Figure 22.3) uses a `locate()` method that returns the position of the first occurrence of a given element in a 1D array of integers; if the element is not present, `-1` is returned.

```

public class LinearSearch {
    // Return the position of the given element; -1 if not there
}

```

Array Size	Time in msec	
	Selection Sort	Quicksort
10	< 1	4
50	1	4
100	3	5
500	15	16
1,000	29	18
5,000	464	31
10,000	1,839	41
50,000	48,642	120
100,000	238,220	206

Table 22.1: Empirical data comparing selection sort to Quicksort. This data was obtained running CompareSorts (Figure 22.2). The results are averaged over five runs.

```

public static int locate(int[] a, int seek) {
    for ( int i = 0; i < a.length; i++ ) {
        if ( a[i] == seek ) {
            return i;    // Return position immediately
        }
    }
    return -1;    // Element not found
}

// Print an integer right-justified in a 4-space field
private static void format(int i) {
    if ( i > 9999 ) {
        System.out.print("****");
        return;
    }
    if ( i < 1000 ) {
        System.out.print(" ");
        if ( i < 100 ) {
            System.out.print(" ");
            if ( i < 10 ) {
                System.out.print(" ");
            }
        }
    }
    System.out.print(i);
}

// Print the contents of the array
public static void print(int[] a) {
    for ( int i : a ) {
        format(i);
    }
    System.out.println();    // newline
}

// Tab over the given number of spaces
private static void tab(int spaces) {

```

```

        for ( int i = 0; i < spaces; i++ ) {
            System.out.print("    ");
        }
    }
    private static void display(int[] a, int value) {
        int position = locate(a, value);
        if ( position >= 0 ) {
            print(a);
            tab(position);
            System.out.println("    ^");
            tab(position);
            System.out.println("    |");
            tab(position);
            System.out.println("    +-- " + value);
        } else {
            System.out.print(value + " not in ");
            print(a);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        int[] list = { 100, 44, 2, 80, 5, 13, 11, 2, 110 };
        display(list, 13);
        display(list, 2);
        display(list, 7);
        display(list, 100);
        display(list, 110);
    }
}

```

Listing 22.3: LinearSearch—finds an element within an array

Running LinearSearch produces

```

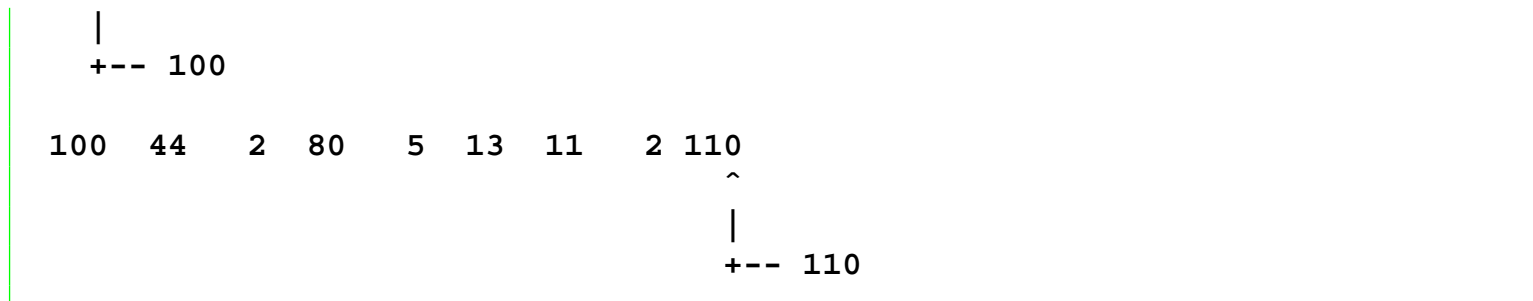
100  44  2  80  5  13  11  2 110
              ^
              |
              +-- 13

100  44  2  80  5  13  11  2 110
          ^
          |
          +-- 2

7 not in 100  44  2  80  5  13  11  2 110

100  44  2  80  5  13  11  2 110
^

```



The key method in the `LinearSearch` class is `locate()`; all the other methods simply lead to a more interesting display of `locate()`'s results. Examining each method reveals:

- `locate()`—The `locate()` method begins at the beginning of the array and compares each element with the item sought.

```

// Return the position of the given element; -1 if not there
public static int locate(int[] a, int seek) {
    for ( int i = 0; i < a.length; i++ ) {
        if ( a[i] == seek ) {
            return i;    // Return position immediately
        }
    }
    return -1;    // Element not found
}
  
```

If a match is found, the position of the matching element is immediately returned; otherwise, if all the elements of the array are considered and no match is found, `-1` is returned (`-1` can never be a legal index in a Java array).

- `format()`—`format()` prints an integer right-justified within a four-space field. Extra spaces pad the beginning of the number if necessary.
- `print()`—`print()` prints out the elements in any integer array using the `format()` method to properly format the values. This alignment simplifies the `display()` method.
- `tab()`—`tab()` prints four blank spaces and leaves the cursor on the same line. It is used by `display()` to position its output.
- `display()`—`display()` uses `locate()`, `print()`, and `tab()` to provide a graphical display of the operation of the `locate()` method.

The kind of search performed by `locate()` is known as *linear* search since a straight line path is taken from the beginning to the end of the array considering each element in order. Figure 22.1 illustrates linear search.

Linear search is acceptable for relatively small arrays, but the process of examining each element in a large array is time consuming. An alternative to linear search is *binary search*. In order to perform binary search an array must be in sorted order. Binary search takes advantage of this organization by using a clever but simple strategy that quickly zeros in on the element to find:

1. Let

- *A* be the array,

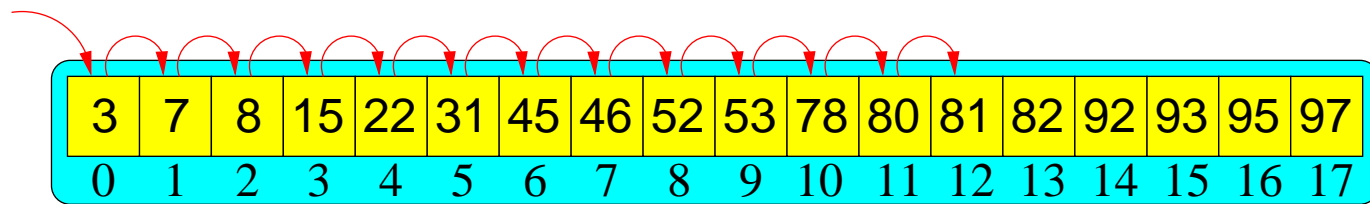


Figure 22.1: Linear search of an array. The algorithm makes 13 probes before value 81 is found to be at index 12.

- n be the length of A , and
 - x be the element to find.
2. Find the middle position, m , of A . $m = \lfloor \frac{n}{2} \rfloor$.
 3. If $A[m] = x$, the element is found and the answer is m ; stop.
 4. If $n = 1$, the element is not present and the answer is -1 ; stop.
 5. Otherwise,
 - if $A[m] > x$, then continue the search using the same strategy with a new array consisting of elements $0 \dots m - 1$ of the current array, or
 - if $A[m] < x$, then continue the search using the same strategy with a new array consisting of elements $m + 1 \dots n - 1$ of the current array.

This approach is analogous to looking for a telephone number in the phone book in this manner:

1. Open the book at its center. If the name of the person is on one of the two visible pages, look at the phone number.
2. If not, and the person's last name is alphabetically less the names of the visible pages, apply the search to the left half of the open book; otherwise, apply the search to the right half of the open book.
3. Discontinue the search with failure if the person's name should be on one of the two visible pages but is not present.

This algorithm can be converted into a Java method:

```
// Return the position of the given element; -1 if not there
public static int binarySearch(int[] a, int seek) {
    int first = 0,           // Initially the first element in array
        last = a.length - 1, // Initially the last element in array
        mid;                 // The middle of the array
    while ( first <= last ) {
        mid = first + (last - first + 1)/2;
        if ( a[mid] == seek ) {
            return mid;      // Found it
        }
        if ( a[mid] > seek ) {
```



```

        last = mid - 1;    // last decreases
    } else {
        first = mid + 1;   // first increases
    }
}
return -1;
}

```

In `binarySearch()`:

- The initializations of `first` and `last`:

```

int first = 0,           // Initially the first element in array
last = a.length - 1,    // Initially the last element in array

```

ensure that $\text{first} \leq \text{last}$ for a nonempty array. If the array is empty, then

$$(\text{first} = 0) > (\text{a.length} - 1 = -1)$$

and the loop body will be skipped. In this case -1 will be returned. This is correct behavior because an empty array cannot possibly contain any item we seek.

- The calculation of `mid` ensures that $\text{first} \leq \text{mid} \leq \text{last}$.
- If `mid` is the location of the sought element (checked in the first `if` statement), the loop terminates.
- The second `if` statement ensures that either `last` decreases or `first` increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually $\text{first} > \text{last}$, and the loop will terminate.
- The second `if` statement also excludes the irrelevant elements from further search.

Figure 22.2 illustrates how binary search works.

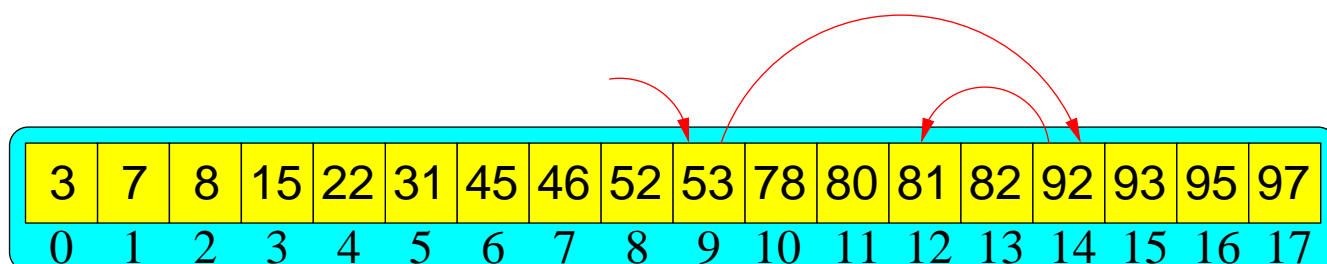


Figure 22.2: Binary search of an array. Compare this algorithm's three probes to locate the value 81 to the 13 probes required by linear search in Figure 22.1.

The Java implementation of the binary search algorithm is more complicated than the simpler linear search algorithm. Ordinarily simpler is better, but for algorithms that process data structures that can potentially hold large amounts of data, more complex algorithms that employ clever tricks that exploit the structure of the data (as binary search does) often dramatically outperform simpler, easier-to-code algorithms. `SearchCompare` (Figure 22.4) compares the performance of linear search versus binary search on arrays of various sizes.

```

import java.util.Random;
import java.util.Arrays;

public class SearchCompare {
    // Return the position of the given element; -1 if not there.
    // Note modification for ordered array---need not always loop to
    // very end.
    public static int linearSearch(int[] a, int seek) {
        for ( int i = 0; i < a.length && a[i] <= seek; i++ ) {
            if ( a[i] == seek ) {
                return i;    // Return position immediately
            }
        }
        return -1;    // Element not found
    }

    // Return the position of the given element; -1 if not there
    public static int binarySearch(int[] a, int seek) {
        int first = 0,           // Initially the first element in array
            last = a.length - 1, // Initially the last element in array
            mid;                  // The middle of the array
        while ( first <= last ) {
            mid = first + (last - first + 1)/2;
            if ( a[mid] == seek ) {
                return mid;    // Found it
            }
            if ( a[mid] > seek ) {
                last = mid - 1; // last decreases
            } else {
                first = mid + 1; // first increases
            }
        }
        return -1;
    }

    // Print the contents of the array
    public static void print(int[] a) {
        for ( int i : a ) {
            System.out.print(i + " ");
        }
        System.out.println(); // newline
    }

    public static void main(String[] args) {
        // Program must have size to be able to run
        if ( args.length != 2 ) {
            System.out.println("Usage:");
            System.out.println("    java SearchCompare <array_size>"
                               + " <iterations>");
            System.exit(1);
        }
        // Allocate the arrays size provided on the command line

```

```

int[] list = new int[Integer.parseInt(args[0])];
// Create a random number generator
Random random = new Random();
// Initialize the array with random integers
for ( int i = 0; i < list.length; i++ ) {
    list[i] = random.nextInt();
}
// Sort the array
Arrays.sort(list);
// print(list);
int linearTime = 0, binaryTime = 0;
Stopwatch timer = new Stopwatch();
// Profile the searches under identical situations
for ( int i = 0; i < Integer.parseInt(args[1]); i++ ) {
    int seek = random.nextInt();
    timer.start();
    linearSearch(list, seek);
    timer.stop();
    linearTime += timer.elapsed();
    timer.start();
    binarySearch(list, seek);
    timer.stop();
    binaryTime += timer.elapsed();
}
// Report results
System.out.println(args[1] + " searches on array of size "
    + args[0] + ":");
System.out.println("Linear search time: " + linearTime
    + " Binary search time: " + binaryTime);
}
}

```

Listing 22.4: SearchCompare—Compares the performance of linear search versus binary search

The SearchCompare program requires two command line arguments:

1. The size of the array
2. The number of searches to perform.

An array of the given size is initialized with random values and then arranged in ascending order. A random search value is generated, and both linear and binary searches are performed with this value and timed. The accumulated times for the given number of searches is displayed at the end. Observe that the same search value is used over the same array by both the linear search and binary search methods to keep the comparison fair.

Note that the linear search method (`linearSearch()`) of SearchCompare has been optimized for ordered arrays to discontinue the search when a value greater than the value sought is encountered. Thus, given an array of size n , on average only $\frac{n}{2}$ comparisons are required to determine that a sought value is *not* present in the array. The original

version always requires n comparisons if the item sought is not present in the array. This optimization for linear search is only possible if the array is ordered.

Table 22.2 lists empirical data derived from running `SearchCompare`. Figure 22.3 plots the results for arrays with up to 1,000 elements.

Array Size	Time (msec) to perform 15,000 random searches	
	Linear Search	Binary Search
10	80	109
20	122	114
50	186	126
100	322	152
200	578	176
500	1394	185
1000	2676	197
5000	13251	227
10000	26579	252
50000	132725	283

Table 22.2: Empirical data comparing linear search to binary search. This data was obtained running `SearchCompare` (Figure 22.4). The number of searches is fixed at 15,000 per run. The results are averaged over five runs. Figure 22.3 plots this data up to array size 1,000.

The empirical results show that for very small arrays (fewer than 20 elements) linear search outperforms binary search. For larger arrays binary search outperforms linear search handily, and the gap between their performances widens dramatically as even larger arrays are considered. Consider an array with 100,000 elements. Binary search would require approximately six seconds to return a result; searching for the same value in the same array using linear search would typically take 36 minutes and 34 seconds!

To better understand these radically different behaviors, we can analyze the methods structurally, counting the operations that must be performed to execute the methods. The search methods are relatively easy to analyze since they do all the work themselves and do not call any other methods. (If other methods were called, we would need to look into those methods and analyze their structure as well.) Operations that we must consider consist of:

- Assignment—copying a value from one variable to another
- Comparison—comparing two values for equality, less than, greater than, etc.
- Arithmetic—adding, subtracting, multiplying, dividing, computing modulus, etc. of two values
- Increment and decrement—incrementing or decrementing a variable
- Array access—accessing an element in an array using the subscript operator

All of these operations each require only a small amount of time to perform. However, when any of these operations are placed within a loop that executes thousands of iterations, the accumulated time to perform the operations can be considerable.

Another key concept in our analysis is *search space*. Search space is the set of elements that must be considered as the search proceeds. In any correctly devised search algorithm the search space decreases as the algorithm progresses until either the element is found or it is determined that the element is not present in the remaining search space.

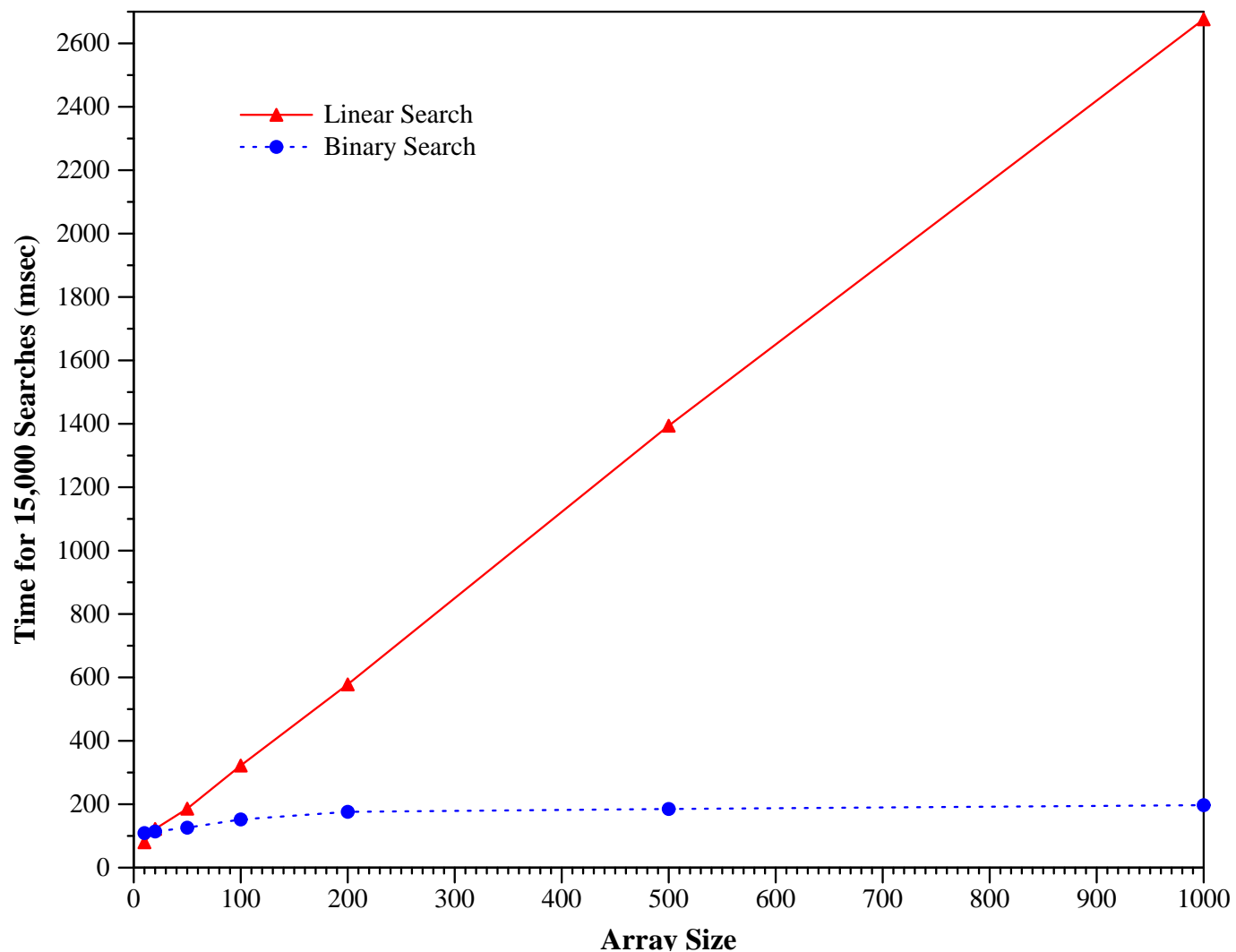


Figure 22.3: Comparison of linear search versus binary search. The results from Table 22.2 for array sizes up to 1,000 are plotted here. The vertical axis represents the number of milliseconds to make 15,000 random searches (averaged over five runs). The horizontal axis represents the number of elements in the array.

The key difference in the performance of linear search versus binary search is the manner in which the search space decreases as each algorithm executes.

Our analysis will be approximate since we are considering the Java source code, but the JVM is actually executing compiled bytecode, not the source code. A single source code expression may result in several bytecode instructions. Also, different types of instructions execute at different rates. For example, usually addition and subtraction operations can be performed faster than multiplication and division operations. For simplicity, we assume all operations require equal time to execute.

Table 22.3 tabulates the number of operations required when executing the `linearSearch()` method.

If the `for` loop is executed n times, then a total of $7n + 2$ operations will be performed by the `linearSearch()` method. The number of loops performed depends on the size of the array being searched. On the average $n = \frac{N}{2}$ for an array of size N . This is because, given a random search value, the following is true:

- If the element is in the array, the loop begins at position zero, and considers each element in turn until the element is located. Each time through the loop the search space decreases by one element. The element is just as likely to be near the front as it is to be near the back. $\frac{N}{2}$ is the “average position” of an element in the array,

Linear Search	Analysis	Effort
for (int i = 0; i < a.length && a[i] <= seek; i++) {	<i>for loop header</i>	
i = 0	Initialization: performed once	1
i < a.length	Comparison: performed each time through loop	n
a[i]	Array access: performed each time through loop	n
a[i] <= seek	Comparison: performed each time through loop	n
i < a.length && a[i] <= seek	Logical <i>and</i> : performed each time through loop	n
i++	Incrementation: performed each time through loop	n
if (a[i] == seek) {	<i>if statement</i>	
a[i]	Array access: performed each time through loop	n
a[i] == seek	Comparison: performed each time through loop	n
return i;	Return: performed at most once	≤ 1
}	<i>End of if</i>	
}	<i>End of for</i>	
return -1;	Return: performed at most once	≤ 1
<i>Total Operations</i>	Exactly one of the return statements will be executed per method call, so both return statements count only as one	$7n + 2$

Table 22.3: A structural analysis of the linear search method. The `for` loop is executed n times.

so $\frac{N}{2}$ iterations must be performed on average to find an element in an ordered array.

- If the element is not in the array, the loop begins at position zero, and considers each element in turn until the array element under consideration is greater than the item sought. Each time though the loop the search space decreases by one element. The “average position” of a missing element is also $\frac{N}{2}$ since the missing element is just as likely to belong in the first part of the array as the last part, so $\frac{N}{2}$ iterations must be performed on average to determine that the element is not present.

Since `linearSearch()` on average performs $\frac{N}{2}$ iterations given an array of size N , it must perform approximately $\frac{7}{2}N + 2$ operations. This is a good measure of the amount of work `linearSearch()` must perform, and thus how long it will take to execute.

Now consider an analysis of the `binarySearch()` method as shown in Table 22.4.

If the `while` loop is executed m times, then a total of $12m + 4$ operations will be performed by the `binarySearch()` method. If n (the number iterations in `linearSearch()`) = m (the number of iterations in `binarySearch()`), then `linearSearch()` will clearly outperform `binarySearch()` for all array sizes. However, $n \neq m$ in general because a binary search traverses an array in a very different manner than linear search. As in the case of linear search, the number of loops performed by `binarySearch()` depends on the size of the array being searched. On the average, for an array of size N , $m = \log_2 N$ because, given a random search value:

Binary Search	Analysis	Effort
<code>first = 0</code>	Assignment: performed once	1
<code>last = a.length - 1</code>	<i>Assignment statement</i>	
<code>a.length - 1</code>	Subtraction: performed once	1
<code>last = a.length - 1</code>	Assignment: performed once	1
<code>while (first <= last) {</code>	<i>while loop header</i>	
<code>first <= last</code>	Comparison: performed each time through loop	m
<code>mid = first + (last - first + 1)/2;</code>	<i>Complex assignment statement</i>	
<code>last - first</code>	Subtraction: performed each time through loop	m
<code>(last - first + 1)</code>	Addition: performed each time through loop	m
<code>(last - first + 1)/2</code>	Division: performed each time through loop	m
<code>first + (last - first + 1)/2</code>	Addition: performed each time through loop	m
<code>mid = first + (last - first + 1)/2</code>	Assignment: performed each time through loop	m
<code>if (a[mid] == seek) {</code>	<i>if statement header</i>	
<code>a[mid]</code>	Array access: performed each time through loop	m
<code>a[mid] == seek</code>	Comparison: performed each time through loop	m
<code>return mid</code>	Return: performed at most once	≤ 1
<code>if (a[mid] > seek) {</code>	<i>if statement header</i>	
<code>a[mid]</code>	Array access: performed each time through loop	m
<code>a[mid] > seek</code>	Comparison: performed each time through loop	m
<i>Exactly one of</i> <code>last = mid - 1</code>	Subtraction and assignment (two operations)	
<i>or</i> <code>first = mid + 1</code>	Addition and assignment (two operations)	
<i>is performed each time through loop</i>		$2m$
<code>return -1</code>	Return: performed at most once	≤ 1
<i>Total Operations</i>	Exactly one of the <code>return</code> statements will be executed per method call, so both <code>return</code> statements count only as one	$12m + 4$

Table 22.4: A structural analysis of the binary search method. The `while` loop is executed m times.

- If the element is in the array, the loop begins at the middle position in the array. If the middle element is the item sought, the search is over. If the sought item is not found, search continues over either the first half or the second half of the remaining array. Thus, each time through the loop the search space decreases by *one half*. At most $\log_2 N$ iterations are performed until the element is found.
- If the element is not in the array, the algorithm probes the middle of a series of arrays that decrease in size by one half each iteration until no more elements remain to be checked. Again, each time through the loop the search space decreases by *one half*. At most $\log_2 N$ iterations are performed until the remaining array is reduced to zero elements without finding the item sought.

Why $\log_2 N$? In this case $\log_2 x$ represents the number of times x can be divided in half (integer division) to obtain a value of one. As examples:

- $\log_2 1024 = 10$: $1,024 \xrightarrow{1} 512 \xrightarrow{2} 256 \xrightarrow{3} 128 \xrightarrow{4} 64 \xrightarrow{5} 32 \xrightarrow{6} 16 \xrightarrow{7} 8 \xrightarrow{8} 4 \xrightarrow{9} 2 \xrightarrow{10} 1$
- $\log_2 400 = 8.6$: $400 \xrightarrow{1} 200 \xrightarrow{2} 100 \xrightarrow{3} 50 \xrightarrow{4} 25 \xrightarrow{5} 12 \xrightarrow{6} 6 \xrightarrow{7} 3 \xrightarrow{8} 1$

For further justification, consider `Log2` (▮22.5) that counts how many times a number can be halved.

```
public class Log2 {
    private static int log_2(int n) {
        int count = 0;
        while ( n > 1 ) {
            n /= 2;
            count++;
        }
        return count;
    }
    public static void main(String[] args) {
        int badCalc = 0,
            goodCalc = 0;
        final int MAX = 100000;
        // Compare halving to log base 2
        for ( int i = 1; i < MAX; i++ ) {
            if ( log_2(i) == (int)(Math.log(i)/Math.log(2)) ) {
                goodCalc++;
            } else {
                badCalc++;
            }
        }
        System.out.println("Agreements: " + goodCalc + "/" + (MAX - 1));
        System.out.println("Discrepancies: " + badCalc + "/" + (MAX - 1));
    }
}
```

Listing 22.5: `Log2`—demonstrates that $\log_2 x$ corresponds to determining how many times x can be divided in half

In `Log2`, the method `log_2()` determines how many times an integer can be divided by 2 until it is reduced to 1. This result is compared to finding the base 2 logarithm of that value. Java's `Math` class does not have a \log_2 method, but a basic mathematics identity shows us that given the logarithm of a particular base we can compute the logarithm of any other positive base:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Thus Java's `Math.log()` method (which is actually \ln , see Section 13.5) is used to compute \log_2 :

$$\frac{\log_e x}{\log_e 2} = \frac{\text{Math.log}(x)}{\text{Math.log}(2)} = \log_2 x$$

The results show total agreement:

Agreements: 99999/99999
Discrepancies: 0/99999

Since `binarySearch()` on average performs $\log_2 N$ iterations given an array of size N , it must perform approximately $12\log_2 N + 4$ operations. As in the case of `linearSearch()` above, this is a good measure of the amount of work `binarySearch()` must perform, and thus how long it will take to execute.

Finally, we can compare our structural analysis to the empirical results. Figure 22.4 plots the mathematical functions over the various array sizes.

Despite our approximations that skew the curves a bit, the shape of the curves in Figure 22.4 agree closely with the empirical results graphed in Figure 22.3. The curves represent the *time complexity* of the algorithms; that is, how an increase in the size of the data set increases the algorithm's execution time. Linear search has a complexity proportional to N , whereas binary search's complexity is proportional to $\log_2 N$. Both graphs show that as the data size grows, the amount of time to perform a linear search increases steeply; linear search is impractical for larger arrays. For an application that must search arrays that are guaranteed to remain small (less than 20 elements), linear search is the better choice. Binary search, however, performs acceptably for smaller arrays and performs exceptionally well for much larger arrays. We say it *scales* well. Observe that both algorithms work correctly; sometimes successful applications require more than correct algorithms.

22.3 Summary

- Add summary items here.

22.4 Exercises

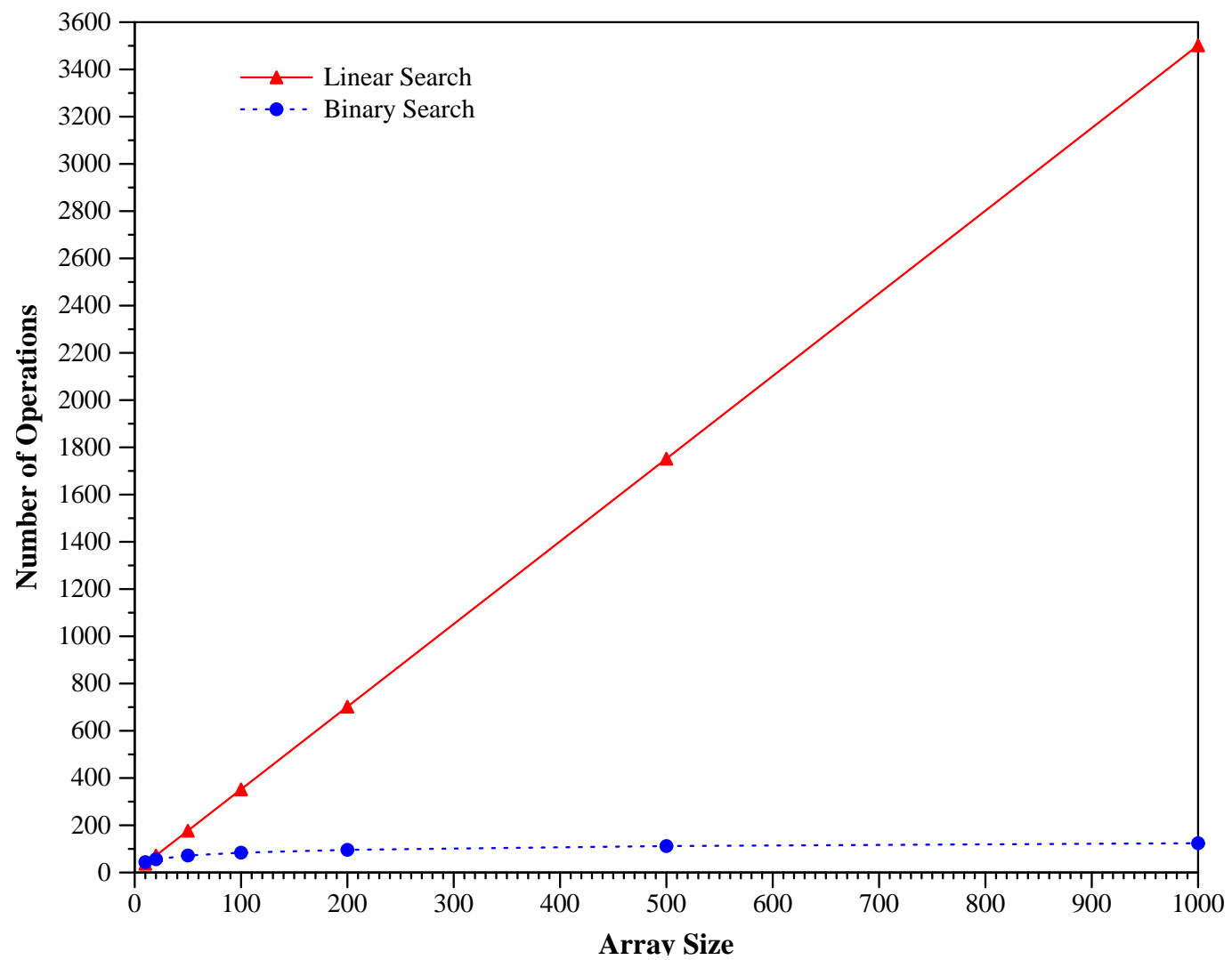



Figure 22.4: Plot of the mathematical functions obtained through the code analysis. The vertical axis represents the number of operations performed. The horizontal axis maps N , the size of the array. The linear search function is $\frac{7}{2}N + 2$, and the binary search function is $12\log_2 N + 4$.

Chapter 23

A Simple Database

We can use an array to implement a simple database of employee records.  23.1 provides a template from which employee record objects can be created.

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class EmployeeRecord {
    private static int ID = 0; // Used to produce consecutive ID numbers
    public static final Scanner scanner = new Scanner(System.in);
    private int idNumber;
    private String lastName;
    private String firstName;
    private int salary;

    public EmployeeRecord(String last, String first, int pay) {
        idNumber = ID++;
        lastName = last;
        firstName = first;
        salary = pay;
    }
    // Accessor methods
    public int getID() { return idNumber; }
    public String getLast() { return lastName; }
    public String getFirst() { return firstName; }
    public int getSalary() { return salary; }

    public void show() {
        DecimalFormat idFmt = new DecimalFormat("00000"),
            salaryFmt = new DecimalFormat("$#####.00");
        System.out.println("ID:      " + idFmt.format(idNumber));
        System.out.println("Name:   " + lastName + ", " + firstName);
        System.out.println("Salary: " + salaryFmt.format(salary));
    }
}
```

```

public static EmployeeRecord read() {
    String last, first;
    System.out.print("Enter last name: ");
    last = scanner.next();
    System.out.print("Enter first name: ");
    first = scanner.next();
    System.out.print("Enter salary: ");
    int pay = scanner.nextInt();
    scanner.nextLine(); // Consume rest of line
    return new EmployeeRecord(last, first, pay);
}
public void edit() {
    String newLast, newFirst, newSalary;
    scanner.nextLine(); // Clear line for new entry
    System.out.print("Last name [" + lastName + "]:");
    newLast = scanner.nextLine();
    System.out.print("First name [" + firstName + "]:");
    newFirst = scanner.nextLine();
    System.out.print("Salary [" + salary + "]:");
    newSalary = scanner.nextLine();
    lastName = (newLast.equals(""))? lastName : newLast;
    firstName = (newFirst.equals(""))? firstName : newFirst;
    salary = (newSalary.equals(""))? salary : Integer.parseInt(newSalary);
}
}

```

Listing 23.1: EmployeeRecord—information about a single employee

EmployeeRecord defines a number of fields and methods:

- **Class variables.** The class variable ID is used to automatically generate consecutive unique identification numbers for each employee record object created. The first ID number assigned will be zero. The constructor is responsible for using and updating this master ID number. The process is identical to that used in the Widget class (Figure 9.2) in Section 9.4.
- The scanner constant, which gets user input, is shared by all EmployeeRecord objects. There is no need for each EmployeeRecord object to have its own object to receive keyboard input. It is public and final, so methods in other classes can use it but not reassign it.
- **Instance variables.** Each employee record object has a unique ID number, a last name, a first name, and a salary. Names and salaries need not be unique.
- **Constructor.** The constructor performs routine object initialization. A unique ID number is assigned outside of the control of the client. The other fields of the new object are assigned from client supplied values.
- **Accessor methods.** The get...() methods allow clients to read any instance variable of the object.
- **I/O methods.** The show() method uses a java.text.DecimalFormat object to format the output of the ID number and salary. The ID number always displays five digits (for example, ID = 5 displays as 00005), and the

salary field appears as currency (for example, salary = 45000 displays as \$45000.00) even though it is stored as an integer. The `show()` method must be an instance method since it accesses instance variables.

The `read()` method allows a user to type in name and salary information from the keyboard. It is a class method because an instance of `EmployeeRecord` is not required for it to work; it creates and returns a new `EmployeeRecord` object.

The `edit()` method allows a user to modify the name fields or salary field of a record. The user is prompted for each field, and the current value is displayed as a default. If the user simply presses the **Enter** key, the empty string ("") is entered. The empty string will not replace an existing field. Notice that the user cannot change the ID number of an employee. This is not allowed because a user may accidentally reassign a used ID number.

A real-world employee record would contain many more fields including date of birth, hire date, social security number, etc.

The database class (Figure 23.2, Database) uses an array to store a collection of employee records.

```
public class Database {
    // Array of employee records
    private EmployeeRecord[] employees;
    // Current number of employees
    private int size;
    public Database(int max) {
        // Create an array to hold all the employees
        employees = new EmployeeRecord[max];
        size = 0; // Currently empty
    }
    public EmployeeRecord retrieve(int id) {
        for ( int i = 0; i < size; i++ ) {
            if ( employees[i].getID() == id ) {
                return employees[i]; // Return record
            }
        }
        return null; // Not found
    }
    public boolean insert(EmployeeRecord rec) {
        if ( size < employees.length ) {
            // Still room to add
            if ( retrieve(rec.getID()) == null ) { // Disallow duplicate IDs
                employees[size++] = rec;
                return true; // Success
            } else {
                System.out.println("Duplicate ID. Record not inserted");
            }
        } else {
            System.out.println("Database full. Record not inserted");
        }
        return false; // Database full or duplicate record; could not insert
    }
    public void show() { // Print all database entries
        for ( int i = 0; i < size; i++ ) {
```

```

        employees[i].show();
        System.out.println("-----");
    }
}

public void sortIDs() { // Sort the database by ascending IDs
    for ( int i = 0; i < size - 1; i++ ) {
        int smallestPos = i;
        for ( int j = i + 1; j < size; j++ ) {
            if ( employees[j].getID() < employees[smallestPos].getID() ) {
                smallestPos = j;
            }
        }
        if ( smallestPos != i ) { // Swap
            EmployeeRecord temp = employees[i];
            employees[i] = employees[smallestPos];
            employees[smallestPos] = temp;
        }
    }
}

public void sortNames() { // Sort the database lexicographically
    for ( int i = 0; i < size - 1; i++ ) {
        int smallestPos = i;
        for ( int j = i + 1; j < size; j++ ) {
            String jName = employees[j].getLast() +
                           employees[j].getFirst(),
                smallestName = employees[smallestPos].getLast() +
                              employees[smallestPos].getFirst();
            if ( jName.compareToIgnoreCase(smallestName) < 0 ) {
                smallestPos = j;
            }
        }
        if ( smallestPos != i ) { // Swap
            EmployeeRecord temp = employees[i];
            employees[i] = employees[smallestPos];
            employees[smallestPos] = temp;
        }
    }
}
}
}

```

Listing 23.2: Database—uses an array to store a collection of employee records

The Database class provides two fields and a number of methods:

- **Instance variables.** An array (`employees`) stores individual employee records. The array's size is allocated when the database is created, but it is initially *logically* empty—it contains no records until they are inserted. The *physical* size of the array is determined by how many elements it can contain; this is fixed when the `new` operator is used to create the array. The *logical* size of the array is based on how many of the allocated spaces

are being used. The `size` variable keeps track of the array's logical size. Each time an element is added, `size` increases by one.

- **Construction.** The constructor creates the array and sets its logical size to zero. The physical size of the array is provided as a parameter from the client.
- **Insertion.** The `insert()` method allows clients to populate the database. The new employee record is placed “on the end” of the array. This means the new record goes in position `size`. The `size` variable is then incremented so that when the `insert()` method is called again the new record will go in the next available position.

Insertion can fail for two reasons:

- The logical size of the array equals its physical size. This means the array is full, and no more elements can be inserted.
- The client attempts to insert a record with an ID number that matches an ID number already present in the database. Duplicate IDs are not allowed, so the insertion is not performed. Because of the way the `EmployeeRecord` class is designed, this situation should never arise; however, a redundant check like this one is not necessarily bad. The author of the `EmployeeRecord` class can be different from the author of the `Database` class. The `Database` programmer is exhibiting *defensive programming*; that is, attempting to avoid an error in the `insert()` method that is actually due to erroneous code elsewhere. This defensive programming is not free, however. The `retrieve()` method is relatively expensive since it will step through the entire array if the element to insert has a unique ID. After thorough testing this conditional check really should be removed.
- **Retrieval.** Employee records are retrieved by ID number. The `retrieve()` method returns a reference to the record (or `null` if not present).
- **Sorting.** The database can be sorted in one of two ways:
 - **ID number order.** This is the “natural” ordering since new records are given sequential ID numbers.
 - **Name order.** This is called lexicographical order (meaning “dictionary order”), commonly called alphabetical order. Since last names are not unique, the first name is concatenated to the last so both are considered when determining the proper order. The `String` method `compareToIgnoreCase()` is oblivious to capitalization and works as follows:

$$a.compareToIgnoreCase(b) \begin{cases} < 0 & \text{if } a \text{ is lexicographically less than } b \\ = 0 & \text{if } a \text{ is equal to } b \\ > 0 & \text{if } a \text{ is lexicographically greater than } b \end{cases}$$

- **Display.** The `show()` method steps through the array and has each employee record object show itself.

This database is extremely rudimentary. A real database would provide for more sophisticated client interaction. Most modern databases provide a query language that allow clients to perform complex queries.

`EmployeeDatabase` (Figure 23.3) represents client code that uses the `Database` and `EmployeeRecord` classes. It allows the user to enter single letter commands and prompts the user when more information is needed.

```
public class EmployeeDatabase {
    public static void main(String[] args) {
        Database db = new Database(10); // Create a new database
        boolean done = false;
        do {
```



```

        System.out.print("==>");
        String command = EmployeeRecord.scanner.nextLine();
        switch ( command.charAt(0) ) {
            case 'i':
            case 'I':
                if ( !db.insert(EmployeeRecord.read()) ) {
                    System.out.println("Record not inserted");
                }
                break;
            case 'e':
            case 'E':
                System.out.print("Enter ID number: ");
                int id = EmployeeRecord.scanner.nextInt();
                EmployeeRecord rec = db.retrieve(id);
                if ( rec != null ) {
                    rec.edit();
                }
                break;
            case 'p':
            case 'P': db.show();    break;
            case 'n':
            case 'N': db.sortNames();    break;
            case 's':
            case 'S': db.sortIDs();    break;
            case 'q':
            case 'Q': done = true; break;
        }
    } while ( !done );
}

```

Listing 23.3: EmployeeDatabase—allows a user to interactively exercise the employee database

23.1 Summary

- Add summary items here.

23.2 Exercises

Chapter 24

Graphical Objects

The simple graphics we saw earlier created *screen artifacts*. A screen artifact is simply an image drawn on the screen (viewport), just as a picture can be drawn on a whiteboard. A picture on the whiteboard can be erased, or another image can be drawn over top of an existing image, but we cannot move an image on a whiteboard unless we erase the existing image and try to draw an exact copy of the original picture in a new location. The “moved” picture is *not* the original picture; it is simply a copy. A picture drawn on a whiteboard is an artifact produced by writing with a marker on the board. The picture cannot exist (outside the mind of the artist, anyway!) without the existence of the whiteboard.

Compare our marker-drawn artifact to a picture drawn on a sticky note (like a 3M Post-it® note). We can stick the note (and its associated picture) anywhere on the whiteboard. If it gets in the way of our current work on the board, we can pull it off and stick it somewhere else without redrawing the picture. The sticky note is an object that can exist independent of the whiteboard. The note can be manipulated in ways that are impossible with an artifact.

A *graphical object* is a software object that can be visualized and manipulated programmatically. Often users can directly interact with graphical objects. Examples of graphical objects include buttons.

24.1 The GraphicalObject Class

Our `GraphicalObject` class provides the basic functionality for graphical objects. To use a graphical object you follow these steps:

1. Create an instance of a subclass of `GraphicalObject`.
2. Add the graphical object instance to an existing viewport.

The subclass of `GraphicalObject` must override the `draw()` method in order for the graphical object to be visible within the viewport. Your subclass constructor must also set the object’s width and height. `SimpleStar` (Figure 24.2) illustrates a simple graphical object that is shaped like a star. When the user clicks

The `GraphicalObject` class has many similarities to our `Viewport` class:

- It has all the methods for drawing primitive graphical shapes: lines, rectangles, polygons, etc. The methods are used just as they are in `Viewport` objects—they are called from within the graphical object’s `draw()` method.
- It provides the same methods for manipulating the size and position: `setSize()` and `setLocation()`.

- It can receive input events. Methods named identically to their viewport counterparts can be overridden to handle mouse activity and key presses.
- Other graphical objects can be added to a graphical object just as a graphical object can be added to a viewport.

GraphicalStarObject (Figure 24.1) defines a simple star-shaped graphical object:

```
import edu.southern.computing.oopj.GraphicalObject;

public class GraphicalStarObject extends GraphicalObject {
    public GraphicalStarObject(int size) {
        super(size, size);
        setCursor(HAND);
        setBackground(TRANSPARENT);
    }
    public void draw() {
        // Code adapted from http://www.research.att.com/
        //                               sw/tools/yoix/doc/graphics/
        //                               pointInPolygon.html
        int size = getWidth(),
            xCenter = size/2,
            yCenter = size/2;
        drawPolygon(xCenter, yCenter - round(0.50*size),
                    xCenter + round(0.29*size), yCenter + round(0.40*size),
                    xCenter - round(0.47*size), yCenter - round(0.15*size),
                    xCenter + round(0.47*size), yCenter - round(0.15*size),
                    xCenter - round(0.29*size), yCenter + round(0.40*size));
    }

    public void setCenter(int x, int y) {
        // Center star at (x,y)
        super.setLocation(x - getWidth()/2, y - getHeight()/2);
    }
}
```

Listing 24.1: GraphicalStarObject—a simple star-shaped graphical object

The `setCenter()` method conveniently allows us to position the star relative to its center instead of relative to the left-top corner of its bounding box. SimpleStar (Figure 24.2) uses the GraphicalStarObject:

```
import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;

public class SimpleStar {
    public static void main(String[] args) {

        final GraphicalStarObject star = new GraphicalStarObject(100);
```

```

// Create a viewport to hold our star
Viewport w = new Viewport("Star: Click to move", 100, 100, 600, 500) {
    public void mouseReleased() { // Reposition the star
        star.setCenter(getMouseX(), getMouseY());
    }
};

w.add(star); // Add the star to the viewport
}

```

Listing 24.2: SimpleStar—uses the star object

Here we override the viewport's mouse released method to reposition the star's center to the location of the mouse event. Notice how the star can now be moved from one place to another within the viewport just as a post it note can be moved from one place on a whiteboard to another. The star object itself is responsible for rendering itself, not the star's client code.

One built-in capability of graphical objects allows for a more interesting effect. All graphical objects can be made movable via a `setMovable()` method that accepts a Boolean value: `true` = movable and `false` = immovable. The user can drag a movable graphical object around the viewport with the mouse. `MovableStar` (Figure 24.3) allows the user to drag the star around in the viewport:

```

import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;

public class MovableStar {
    public static void main(String[] args) {

        final GraphicalStarObject star = new GraphicalStarObject(100);
        star.setMovable(true);

        // Create a viewport to hold our star
        Viewport w = new Viewport("Star: Click to move", 100, 100, 600, 500);

        w.add(star);

    }
}

```

Listing 24.3: MovableStar—allows the user to drag the star around in the viewport

To illustrate how graphical objects can be composed, see `MovableCompositeStar` (Figure 24.4):

```

import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;

```

```

public class MovableCompositeStar {
    public static void main(String[] args) {

        // Make a small, movable star object
        GraphicalStarObject littleStar = new GraphicalStarObject(20);
        littleStar.setMovable(true);

        // Make a bigger star object
        GraphicalStarObject bigStar = new GraphicalStarObject(100);
        // Add the little star to the big star
        bigStar.add(littleStar);
        littleStar.setCenter(bigStar.getWidth()/2, bigStar.getHeight()/2);
        bigStar.setMovable(true);

        // Create a viewport to hold our star
        Viewport w = new Viewport("Composite Stars", 100, 100, 600, 500);

        // Add the big star to the viewport
        w.add(bigStar);
    }
}

```

Listing 24.4: MovableCompositeStar—graphical objects can be composed of other graphical objects

Here we add the little star, a graphical object in its own right, to another graphical object, the big star. Notice how the big star can be dragged taking the little star with it, and the little star can be moved within the big star without affecting the position of the big star.

ContextMenu objects can be added to graphical objects. This is handy when you want the user to change an object's property. In ChangableStar (Figure 24.5), the user can change a star's color and size:

```

import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;
import edu.southern.computing.oopj.GraphicalText;
import edu.southern.computing.oopj.ContextMenu;

public class ChangableStar {
    public static void main(String[] args) {

        // Create a viewport to hold our stars
        Viewport w = new Viewport("Changable Stars", 100, 100, 600, 500);

        GraphicalStarObject star1 = new GraphicalStarObject(100);
        star1.setMovable(true);
        star1.setContextMenu(new StarMenu(star1));
        star1.setLocation(100, 100);
        w.add(star1);
    }
}

```

```

        GraphicalStarObject star2 = new GraphicalStarObject(100);
        star2.setMovable(true);
        star2.setContextMenu(new StarMenu(star2));
        star2.setLocation(200, 100);
        w.add(star2);
    }
}

```

Listing 24.5: ChangableStar—graphical objects with popup menus

The `StarMenu` type is a subclass of `ContextMenu`, as shown in `StarMenu` (Figure 24.6):

```

import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;
import edu.southern.computing.oopj.GraphicalText;
import edu.southern.computing.oopj.ContextMenu;

public class StarMenu extends ContextMenu {
    private GraphicalStarObject star;

    public StarMenu(GraphicalStarObject star) {
        super("Red", "Blue", "Black", "Bigger", "Smaller", "Quit");
        this.star = star;
    }

    private void scale(double factor) {
        // Current size
        int size = star.getWidth();
        // There is a limit to the scaling
        if (size > 5 && size < 500) {
            // Compute star's current center
            int xCenter = star.getX() + size/2,
                yCenter = star.getY() + size/2;
            // Resize the star
            star.setSize(GraphicalObject.round(factor * size),
                GraphicalObject.round(factor * size));
            // Recenter the scaled star (according to new size)
            star.setLocation(xCenter - star.getWidth()/2,
                yCenter - star.getHeight()/2);
        }
    }

    public void handler(String item) {
        if (item.equals("Red")) {
            star.setForeground(GraphicalObject.RED);
        } else if (item.equals("Blue")) {

```

```

        star.setForeground(GraphicalObject.BLUE);
    } else if (item.equals("Black")) {
        star.setForeground(GraphicalObject.BLACK);
    } else if (item.equals("Bigger")) {
        scale(1.2);
    } else if (item.equals("Smaller")) {
        scale(0.8);
    } else if (item.equals("Quit")) {
        System.exit(0);
    }
    star.visuallyUpdate();
}
}

```

Listing 24.6: StarMenu—a context-sensitive popup menu for modifying GraphicalStarObjects

24.2 Graphical Text Objects

GraphicalText is a straightforward extension (that is, subclass) of GraphicalObject. Its constructor accepts a string argument, and a graphical text object renders itself by drawing the string. TextGraphics (Figure 24.7) adds some graphical text objects to a viewport:

```

import edu.southern.computing.oopj.Viewport;
import edu.southern.computing.oopj.GraphicalObject;
import edu.southern.computing.oopj.GraphicalText;

public class TextGraphics {
    public static void main(String[] args) {
        // Create a white viewport to hold our text
        Viewport w = new Viewport("Text Graphics", 100, 100, 600, 500);
        w.setBackground(Viewport.WHITE);

        // Create and set up the text objects
        GraphicalText text1 = new GraphicalText("String #1: Can't move me");
        text1.setLocation(10, 10);

        GraphicalText text2 = new GraphicalText("String #2: CAN move me");
        text2.setMovable(true);
        text2.setLocation(10, 30);

        GraphicalText text3 = new GraphicalText("String #3: I'm red!");
        text3.setForeground(GraphicalText.RED);
        text3.setLocation(10, 70);

        GraphicalText text4 = new GraphicalText("String #4: I'm blue!");
        text4.setForeground(GraphicalText.BLUE);
        text4.setMovable(true);
    }
}

```

```
        text4.setLocation(10, 110);

        // Add the text to the viewport
        w.add(text1);
        w.add(text2);
        w.add(text3);
        w.add(text4);
    }
}
```

Listing 24.7: TextGraphics—uses some movable and immobile graphical text objects

Text objects `text2` and `text4` can be moved by the user; `text1` and `text3` have fixed positions.

24.3 Graphical Buttons

A natural extension of `GraphicalText` is the `GraphicalButton` class. Wrap the graphical text in a box that responds to mouse clicks, and you have a graphical button.

24.4 Summary

- Add summary items here.

24.5 Exercises

1. Add exercises here.

Chapter 25

Exceptions

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve the problem in the general case, but there may be a number of special cases that must all be addressed within the algorithm for the algorithm to be correct. Some of these special cases might occur rarely under the most extraordinary circumstances. For the algorithm to be robust, these exceptional cases must be handled properly; however, adding the necessary details to the algorithm may render it overly complex and difficult to construct correctly. Such an overly complex algorithm would be difficult for others to read and understand, and it would be harder to debug and extend.

Ideally, a developer would write the algorithm in its general form including any common special cases. Exceptional situations that should rarely arise along with a strategy to handle them could appear elsewhere, perhaps as an annotation to the algorithm. Thus, the algorithm is kept focused on solving the problem at hand, and measures to deal with exceptional cases are handled elsewhere.

Java's exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

An *exception* is a special object that can be created when an extraordinary situation is encountered. Such a situation almost always represents a problem, usually some sort of runtime error. Examples of exceptional situations include:

- Attempting to read past the end of a file
- Evaluating the expression `A[i]` where $i \geq A.length$
- Attempting to remove an element from an empty list
- Attempting to read data from the network when the connection is lost (perhaps due to a server crash or the wire being unplugged from the port).

Many of these potential problems can be handled by the algorithm itself. For example, an `if` statement can test to see if an array index is within the bounds of the array. However, if the array is accessed at many different places within a method, the large number of conditionals in place to ensure the array access safety can quickly obscure the overall logic of the method. Other problems such as the network connection problem are less straightforward to address directly in the algorithm. Fortunately, specific Java exceptions are available to cover each of the above problems.

Exceptions represent a standard way to deal with runtime errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. One common approach is for methods to return an integer that represents that method's success. This kind of error handling has its limitations, however. The primary purpose of some methods is to return an integer result that is not an indication of an error (for example, `Integer.parseInt()`). Perhaps a `String` could be returned instead? Unfortunately, some methods naturally return `Strings` (like `next()` in the `Scanner` class). Also, returning a `String` would not work for a method that naturally returns an integer as its result. A completely different type of exception handling technique would need to be developed for methods such as these.

Other error handling strategies are possible. The main problem with these ad hoc approaches to exception handling is that the error handling facilities developed by one programmer may be incompatible with those used by another. Another weakness of programmer-devised error detection and recovery facilities is that the compiler cannot ensure that they are being used consistently or even at all. A comprehensive, uniform exception handling mechanism is needed. Java's exceptions provide such a framework. Java's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. Exceptions are used in the standard Java API, and programmers can create new exceptions that address issues specific to their particular problems. These exceptions all use a common mechanism and are completely compatible with each other. Also, exceptions are part of the Java language specification, and so the compiler has been made to properly enforce the proper use of exceptions.

25.1 Exception Example

A simple example introduces Java's exception handling features. This example uses the following classes and client program:

- `Collection.Collection` (Figure 25.3),
- `Collection.CollectionFullException` (Figure 25.2),
- `Collection.InvalidCollectionSizeException` (Figure 25.1), and
- `CollectTest` (Figure 25.4).

The client code creates and uses a simple custom data structure called a *collection*. A collection has the following characteristics:

- It has three private instance variables:
 - an array of `Object` types; therefore, any reference (that is, nonprimitive) type can be stored in the collection
 - an integer representing a programmer-specified fixed maximum capacity of the collection
 - an integer keeping track of the current number of valid elements in the collection
- It has three public methods:
 - `insert()` allows client code to add elements to the collection
 - `toString()` (overridden method inherited from `java.lang.Object`) renders the contents of the collection in a form suitable for display
 - `isEmpty()` returns true if the collection contains no elements; otherwise, it returns false
 - `isFull()` returns true if the collection has no more room to insert more elements; otherwise, it returns false

- Its constructor accepts a single integer parameter indicating the collection's maximum capacity; initially a collection is empty

Client code can check to see if a collection is empty before trying to print it (although printing an empty collection is not harmful). Client code *should* always check to see that the collection is full before attempting to insert a new element. What if the client programmer fails to include the check for full before insertion? Without exceptions, an array access out of bounds runtime error occurs that terminates the program.

Should a programmer be allowed to create a collection with a zero or negative capacity? Such a collection is useless, so this should be disallowed.

Exceptions can be used to address the above two issues. Two custom exception classes are devised:

- `Collection.InvalidCollectionSizeException` (Figure 25.1) is used when client code attempts to create a collection instance of nonpositive capacity and
- `Collection.CollectionFullException` (Figure 25.2) is used when client code attempts to insert an element into a full collection.

The new exception classes are very simple:

```
package Collection;

public class InvalidCollectionSizeException extends Exception {}
```

Listing 25.1: `Collection.InvalidCollectionSizeException`—Thrown when creating a `Collection` with a nonpositive size

```
package Collection;

public class CollectionFullException extends Exception {}
```

Listing 25.2: `Collection.CollectionFullException`—Thrown when attempting to insert an element into a full `Collection`

Both exceptions extend class `Exception` but provide no additional functionality. This is not unusual as the main purpose of these definitions is to create new *types*. Notice that both `InvalidCollectionSizeException` and `CollectionFullException` are located in the `Collection` package.

Now that the exception classes have been examined, the `Collection.Collection` class (Figure 25.3) itself can be considered.

```
package Collection;
```

```

public class Collection {
    protected Object[] list;    // Array of elements in the collection
    protected int currentSize;  // Current number of viable elements
    protected int maxSize;      // Maximum number of elements
    public Collection(int size) throws InvalidCollectionSizeException {
        if ( size > 0 ) {
            list = new Object[size];    // Allocate collection to maximum size
            maxSize = size;              // Remember maximum size
            currentSize = 0;             // Collection is initially empty
        } else {
            throw new InvalidCollectionSizeException();
        }
    }
    public void insert(Object newElement) throws CollectionFullException {
        if ( currentSize < maxSize ) {
            list[currentSize++] = newElement;
        } else {
            throw new CollectionFullException();
        }
    }
    public boolean isEmpty() {
        return currentSize == 0;
    }
    public boolean isFull() {
        return currentSize == maxSize;
    }
    public String toString() {
        String result = "[";
        for ( int i = 0; i < currentSize; i++ ) {
            result += list[i] + " ";
        }
        for ( int i = currentSize; i < maxSize; i++ ) {
            result += "- ";
        }
        return result + "]";
    }
}

```

Listing 25.3: Collection.Collection—A simple data structure that uses exceptions

Some notable features of the Collection class include:

- The Collection class is part of the Collection package, as are the custom collection exception classes.
- The constructor and insert() method have an additional specification between their parameter lists and bodies:

```

public Collection(int size) throws InvalidCollectionSizeException {

```

```
    . . .
}
```

and

```
public void insert(Object newElement) throws CollectionFullException {
    . . .
}
```

This *exception specification* indicates the types of exceptions that the method or constructor has the potential to *throw*. It specifies that the method can create a new instance of an exception object and pass it (throw it) up to client code that called the method. This has several important consequences to client code that uses such a method:

- Client code is thus warned that the method can throw the indicated type of exception.
- Client code can *catch* the exception object and deal with the problem in some reasonable way. The client code is said to *handle the exception*.
- Client code cannot ignore the possibility of the exception being thrown by the method; client code must do one of two things:
 - * handle the exception itself or
 - * declare the same exception type in its own exception specification (this basically means that the client code is not handling the method's exception but passing the exception up to the code that called the client code)

It is a compile-time error if client code calls a method with an exception specification but does not address the exception in some way.

The exception specification lists all exceptions that the method can throw. Commas are used to separate exception types when a method can throw more than one type of exception:

```
public void f(int a)
    throws ProtocolException, FileNotFoundException, EOFException {
    // Details omitted . . .
}
```

Here, method `f()` has the potential to throw three types of exceptions. Any code using `f()` must ensure that these types of exceptions will be properly handled.

- The `throw` keyword is used to force an exception. Ordinarily a new exception object is created, then it is *thrown*. The act of throwing an exception causes the execution of the code within the method to be immediately terminated and control is transferred to the “closest” exception handler. “Closest” here means the method that is closest in the chain of method calls. For example, if method `main()` calls method `A()` which calls method `B()` which finally calls method `C()`, the call chain looks like

`main() → A() → B() → C()`

If method `C()` throws an exception and method `A()` has the code to handle that exception, `A()` is the closest method up the call chain that can handle the exception. In this case when `C()` throws the exception:

1. `C()`'s execution is interrupted by the throw.
2. Normally control would return from method `C()` back to method `B()` (this is the normal method call return path), but the throw statement causes the return to `B()` to be bypassed.

3. Any code following the call to `B()` within `A()`'s normal program flow is ignored and the exception handling code within `A()` is immediately executed.

Finally, `CollectTest` (Figure 25.4) shows how client code deals with exceptions.

```
import Collection.*;
import java.util.Scanner;

public class CollectTest {
    private static void help() {
        System.out.println("Commands:");
        System.out.println("    n <size>  Create a "
            + "new collection");
        System.out.println("    i <item>  Insert new "
            + "element");
        System.out.println("    p          Print "
            + "contents");
        System.out.println("    h          Show this "
            + "help screen");
        System.out.println("    q          Quit ");
    }
    public static void main(String[] args) {
        Collection col = null;
        // Let the user interact with the collection
        boolean done = false;
        Scanner scan = new Scanner(System.in);
        do {
            System.out.print("Command?: ");
            String input = scan.nextLine();
            try {
                switch (input.charAt(0)) {
                    case 'N':
                    case 'n':
                        // Create a new collection with a new size
                        col = new Collection
                            (Integer.parseInt(input.substring(1).trim()));
                        break;
                    case 'I':
                    case 'i':
                        col.insert(input.substring(1).trim());
                        break;
                    case 'P':
                    case 'p':
                        System.out.println(col);
                        break;
                    case 'H':
                    case 'h':
                        help();
                        break;
                    case 'Q':
```

```

        case 'q':
            done = true;
            break;
    }
} catch ( CollectionFullException ex ) {
    System.out.println("Collection full; nothing "
        + "inserted");
} catch ( InvalidCollectionSizeException ex ) {
    System.out.println("Collection size must be positive");
    System.exit(0);
} catch ( NullPointerException ex ) {
    System.out.println("Use N command to make a new collection");
}
} while ( !done );
}
}

```

Listing 25.4: CollectTest—Exercises the Collection class

The expression

```
input.substring(1).trim()
```

returns a new string based on the original string input:

1. The call to `substring()` returns a substring of input. In this case the substring contains all the characters in input beginning at position 1 (which is the second character). Thus, if input was "abcdef", then `input.substring(1)` would be "bcdef".
2. The call to `trim()` removes all the leading and trailing whitespace from a string. For example, if string `s` is " abc ", then `s.trim()` returns "abc".

If the user types in the string "i fred", then

1. input is "i fred"
2. `input.substring(1)` is " fred"
3. `input.substring(1).trim()` is "fred"

Thus the first letter of the string (`input.charAt(0)`) is used to select the command, and `input.substring(1).trim()` is used if necessary to determine the rest of the command—how big to make the collection or what to insert.

`CollectTest` allows users to enter commands that manipulate a collection. When running the program, pressing H displays a help menu:

```

Commands:
  n <size>  Create a new collection

```

```

i <item>  Insert new element
p          Print contents
h          Show this help screen
q          Quit

```

A loop processes commands from the user until the Q command terminates the program. A sample session looks like:

```

Command?: n 5
Command?: p
[ - - - - ]
Command?: i 10
Command?: p
[ 10 - - - ]
Command?: i 20
Command?: p
[ 10 20 - - ]
Command?: i fred
Command?: p
[ 10 20 fred - - ]
Command?: q

```

What are some potential problems to the user?

- Attempting to insert an item into a full collection
- Attempting to create a collection with a nonpositive capacity
- Attempting to insert an item into a collection before the collection has been created

Each of these problems should raise an exception and the client code must properly handle them should they arise.

The client-side issues of exceptions in `CollectTest` include:

- Code that has the potential of throwing an exception is “wrapped” by a `try` block. Almost all the code within `main()` is placed within `try { . . . }`. There are three potential exceptions `main()` must be prepared to handle:
 - `insert()` invoked by the I command can throw a `CollectionFullException` exception,
 - The collection creation statement invoked by the N command can throw an `InvalidCollectionSizeException` exception,
 - any qualified access using `col` can throw a `NullPointerException` if `col` is null; a qualified access is one in which `col` appears to the left of the dot (`.`) operator, as in


```
col.insert(input.substring(1).trim());
```

Notice that two of these exceptions are our custom exceptions, and one is a standard exception that we’ve seen before (See Section 7.4).

- Three `catch` clauses follow the `try` block. Each `catch` clause provides the code to mitigate a particular type of exception. In this case each remediation simply shows a message to the user, and program execution continues.

If the `try` block (and associated `catch` clauses) were omitted, the compiler would generate an error at the call of `insert()` and the new collection creation. This is because both the `insert()` method and the constructor declare that they can throw an exception in their exception specifications, but the client code would be ignoring this possibility.

25.2 Checked and Unchecked Exceptions

The `try` block could be omitted in `CollectTest` if only the `NullPointerException` were involved because `NullPointerException` is different from the other two exceptions. Its direct superclass is `RuntimeException`. All subclasses of `RuntimeException` are *unchecked exceptions*. The name `RuntimeException` and the term *unchecked exception* are a bit confusing:

- The name `RuntimeException` is an unfortunate choice, since *all* exceptions are generated at run time. The name comes from the fact that in general error checking can be performed at two separate times: compile time and run time. The compiler makes sure that the rules of the language are not violated (example errors include syntax errors and using an uninitialized variable). It can also check to see if provisions have been made to catch potential exceptions. The compiler can do this because some methods provide exception specifications. The compiler can verify that the invocation of a method with an exception specification appears only within a `try` block that provides a `catch` clause for that exception type. The only exception to this rule: exception types derived from `RuntimeException` are *not* checked at compile time.

Like all exceptions, they arise at run time, so they are checked at run time only. Thus their name, `RuntimeException`.

- The term *unchecked exception* does not mean that the JRE does not check for these exceptions at run time. It means the compiler does not check to see if code that can throw exceptions of this type properly handles such exceptions. A method that can throw an instance of a subclass of `RuntimeException` is not required to declare that fact in its exception specification. Even if a method does declare it throws a subclass of `RuntimeException`, client code is not required to catch it. “Unchecked” thus means unchecked at compile time.

In contrast to unchecked exceptions, *checked exceptions*, derived from `Exception`, *are* checked at compile time. A method that can throw a checked exception *must* declare so in its exception specification. Client code that invokes a method that can throw a checked exception, such as

```
void f() throws E {
    // Details omitted
}
```

must do one of two things:

1. it can wrap the invocation in a `try` block and catch that exception type

```
void m() {
    try {
        f();
    } catch ( E e ) {}
}
```

or

2. it can defer the exception handling to its own caller.

```
void m() throws E {  
    f();  
}
```

Option 2 requires the client method to include the deferred exception type in its own exception specification.

`RuntimeException` subclasses represent common runtime errors in many programming language environments: out-of-bounds array access, division by zero, dereferencing a null reference variable, etc. Many of the `RuntimeException` errors do not arise from explicit method calls; they instead involve Java operators: subscript (`[]`), division (`/`), and dot (`.`).

The hierarchy of `Exception` classes is shown in Figure 25.1.

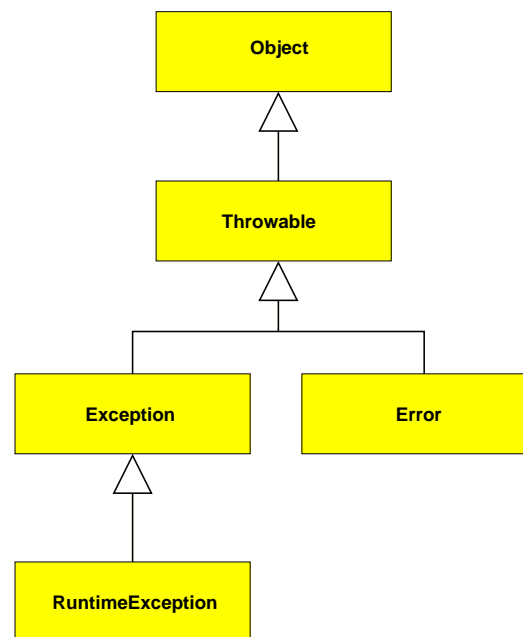


Figure 25.1: The hierarchy of `Exception` classes. These include:

- **Throwable.** The `Throwable` class is the superclass for all exception and error classes. It specifies the methods common to all exception and error classes.
- **Exception.** Programmers normally subclass the `Exception` class when creating custom exceptions. Any method that can throw an instance of a subclass of `Exception` must declare so in its exception specification. This means that client code must be prepared to handle the exception should it arise.
- **Error.** The `Error` class (and any programmer-defined subclasses) represent serious errors that clients are not expected to catch and handle. Programs should unconditionally terminate when an `Error` object is thrown.
- **RuntimeException.** This class and any programmer-defined subclasses represent unchecked exceptions. A method that can throw a `RuntimeException` or any of its subclasses is not required to declare so in its exception specification. Client code is not required to handle `RuntimeExceptions`. `RuntimeExceptions` may be caught by client code, but uncaught `RuntimeExceptions` will terminate the program.

Since most programmer-defined exceptions are derived from the `Exception` class, our focus here is on checked exceptions. The use of checked exceptions leads to more reliable code since the compiler demands that provision be made to handle them. While custom `RuntimeException` subclasses can also be created by programmers, such unchecked exceptions are not nearly as helpful to developers. Such unchecked exceptions may never arise during testing, but show up (by causing the program to terminate with an error message and stack trace) after the software is deployed. This is because client code is not required to catch them and client programmers have forgotten to catch them (or believed they never needed to catch them!). The exceptions may only arise due to unusual circumstances that were not modeled in any of the test cases. Checked exceptions are designed to avoid such problems and should be used instead of unchecked exceptions wherever possible.

25.3 Using Exceptions

Exceptions should be reserved for uncommon errors. For example, the following code adds up all the elements in an integer array named `a`:

```
int sum = 0;
for ( int i = 0; i < a.length; i++ ) {
    sum += a[i];
}
System.out.println("Sum = " + sum);
```

This loop is fairly typical. Another approach uses exceptions:

```
sum = 0;
int i = 0;
try {
    while ( true ) {
        sum += a[i++];
    }
} catch ( ArrayIndexOutOfBoundsException ex ) {}
System.out.println("Sum = " + sum);
```

Both approaches compute the same result. In the second approach the loop is terminated when the array access is out of bounds. The statement is interrupted in midstream so `sum`'s value is not incorrectly incremented. However, the second approach *always* throws and catches an exception. The exception is definitely *not* an uncommon occurrence. Exceptions should not be used to dictate normal logical flow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especially when an exception is thrown. (On one system the exception version was about 50 times slower than the exception-less version.) This overhead is reasonable when exceptions are rare but not when exceptions are part of the program's normal execution.

Exceptions are valuable aids for careless or novice programmers. A careful programmer ensures that code accessing an array does not exceed the array's bounds. Another programmer's code may accidentally attempt to access `a[a.length]`. A novice may believe `a[a.length]` is a valid element. Since no programmer is perfect, exceptions provide a nice safety net. In the `CollectTest` program (▮25.4), a prudent programmer would rewrite the insert case of `switch`:

```
case 'I':
case 'i':
    col.insert(input.substring(1).trim());
    break;
```

as

```
case 'I':
case 'i':
    if ( !col.isFull() ) {
        col.insert(input.substring(1).trim());
    }
    break;
```

This version would avoid the `CollectionFullException`. Since this check may be forgotten, however, the `CollectionFullException` provides the appropriate safety net. Since `CollectionFullException` is a checked exception, the catch clause must be provided, and the programmer is reminded to think about how the exception could arise.

Sometimes it is not clear when an exception is appropriate. Consider adding a method to `Collection` (▮25.3) that returns the position of an element within a collection. The straightforward approach that does not use exceptions could be written:

```
public int find(Object obj) {
    for ( int i = 0; i < currentSize; i++ ) {
        if ( list[i].equals(obj) ) {
            return i; // Found it at position i
        }
    }
    return -1; // Element not present
}
```

Here a return value of -1 indicates that the element sought is not present in the collection. Should an exception be thrown if the element is not present? The following code illustrates:

```
public int find(Object obj)
    throws CollectionElementNotPresentException {
    for ( int i = 0; i < currentSize; i++ ) {
        if ( list[i].equals(obj) ) {
            return i; // Found it at position i
        }
    }
    // Element not there; throw an exception
    throw new CollectionElementNotPresentException();
}
```

In the first approach, an unwary programmer may not check the result and blindly use -1 as a valid position. The exception code would not allow this to happen. However, the first approach is useful for determining *if* an element is present in the collection. If `find(x)` returns -1 , then `x` is not in the collection; otherwise, it is in the collection. If the exception approach is used, a client programmer cannot determine if an element is present without the risk of throwing an exception. Since exceptions should be rare, the second approach appears to be less than ideal. In sum,

- The first approach is more useful, but clients need to remember to properly check the result.
- The second approach provides an exception safety net, but an exception always will be thrown when searching for missing elements.

Which approach is ultimately better? The first version uses a common programming idiom and is the better approach for most programming situations. The exception version is a poorer choice since it is not uncommon to look for an element missing from the collection; exceptions should be reserved for uncommon error situations.

As you develop more sophisticated classes you will find exceptions more compelling. You should analyze your classes and methods carefully to determine their limitations. Exceptions can be valuable for covering these limitations. Exceptions are used extensively throughout the Java standard class library. Programs that make use of these classes must properly handle the exceptions they can throw.

25.4 Summary

- Add summary items here.

25.5 Exercises

1. Add exercises here.

Bibliography

- [1] Eric Gamma and Kent Beck. Junit test infected: Programmers love writing tests. *Java Report*, 3(7), 1998.
- [2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 2000.
- [3] Donald Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley, Reading, Massachusetts, 1970.

Index

Error class, 306
 RuntimeException class, 306

absolute value, 229
 abstract class, 188
 actual parameters, 42
 alias, 79
 anonymous array, 254
 anonymous inner class, 149
 applets, 7
 argument, 38
 array, 241
 array access, 244
 array creation, 241
 array declaration, 241
 array index, 244
 array initialization list, 242
 array length, 246
 array parameter, 253
 array return value, 258
 array search, 268
 array size, 246
 array sort, 265
 array subscript, 244
 assert statement, 177
 assignment operator, 18
 assignment statement, 18
 attributes, 38
 autoboxing, 163

base class, 124
 binary operators, 26
 binary search, 271
 block, 214
 Boolean, 17
 boxing, 163
 browser plug-in, 5
 bug, 91
 bytecode, 6

caller, 37
 can do, 201

catch, 301
 chained assignment, 206
 checked exceptions, 305
 child class, 124
 class, 33
 class constant, 98
 class hierarchies, 187
 class methods, 100
 class variable, 98
 client, 5, 37, 39
 client code, 39
 code conventions, 86
 code reuse, 117
 command line, 259
 command shell, 259
 comment, 39
 compile-time error, 87
 compiler, 2–4
 compiler error messages, interpreting, 87
 component, 117
 component-based software development, 117, 155
 components, 112
 composed of, 115
 compound Boolean expression, 47
 compound statement, 49
 concatenation, 29
 concrete class, 188
 conditional operator, 229
 constants, 20
 constructor, 67
 contradiction, 64
 coverage, 4

 debugger, 4
 declaration statement, 19
 default constructor, 70
 defensive programming, 73, 288
 derived class, 124
 division by zero exception, 90
 do/while statement, 230

 eager test, 184

- editor, 4
- empty statement, 51
- emulation, 6
- encapsulation, 95
- error, 87
- error, compile-time, 87
- error, logic, 90
- error, runtime, 90
- error, syntax, 87
- event, 146
- event driven, 140
- exception, 297
- Exception class, 306
- Exception class hierarchy, 306
- exception specification, 301
- exception, division by zero, 90
- explicit cast, 22
- expressions, 26
- field, 38
- for statement, 232
- for/each statement, 247
- formal parameter, 42
- graphic, 140
- graphical object, 290
- graphical user interface, 140
- GUI, 140
- handling an exception, 301
- has a relationship, 132
- identifier, 22
- if statement, 49
- immutable, 102
- immutable objects, 160
- inclusive or, 47
- index, array, 244
- inheritance, 123
- initialization list, array, 242
- inner class, 148
- instance methods, 100
- instance variable, 38
- instrumenting code, 217
- integer division, 90
- interface, 199
- is a, 126
- iteration, 204
- jagged table, 250
- Java plug-in, 7
- Java Runtime Environment, 5
- Java Virtual Machine, 5
- JRE, 5
- JUnit, 179
- JVM, 5
- keyword, 23
- length of an array, 246
- lexicographical order, 265
- linear search, 271
- linear sequence, 190
- local variables, 41
- logic error, 90
- machine language, 2
- matrix, 249
- medium, 1
- memory management, 6
- method, 34, 38
- method body, 35
- method declaration, 34
- method override, 125
- model, 109
- modular design, 117
- multidimensional array, 249
- multiple inheritance, 137
- nested conditionals, 56
- nested loop, 207
- non-modular, 117
- object, 36
- object-oriented language, 33
- objects, 33
- operations, 38
- operators, 26
- outer class, 148
- overloaded, 30, 70
- overriding a method, 125
- packages, 156
- parameter, 38
- parameter list, 35
- parent class, 124
- picture element, 140
- pixel, 140
- plug-in, browser, 5
- polymorphism, 125, 134
- predicate, 46
- prime number, 215
- primitive types, 33

profiler, 4
profiling, 217
protected, 192
pseudorandom, 166

Quicksort, 267

random numbers, 166
rational number, 74
read only, 20
read-only, 69
recursive, 76
reference variable, 77
regression testing, 184
Relational operators, 46
reserved word, 23
runtime error, 90
RuntimeException, 305

scalar, 239
scale, 282
screen artifacts, 290
search space, 276
security, JRE, 6
selection sort, 265
server, 5
singleton, 193
software, 1
sort, array, 265
source code, 3
source code formatting, 85
standard out, 82
stateful, 68
stateless, 66
statements, 17
string, 17
string literal, 158
style guidelines, 86
subclass, 124
subclassing, 123
subscript, array, 244
superclass, 124
switch statement, 225
syntax error, 87

transitivity of inheritance, 171
tautology, 64
test-driven development, 177
testing, 177
Throwable class, 306
time complexity, 282

transitive, 171
two-dimensional array, 249

unary, 29
unboxing, 163
unchecked exception, 305
unchecked exceptions, 305
Unit testing, 178

value, 15
varargs, 262
variables, 2
view, 109

while statement, 203
wrapper, 161

zero-like values, 242