Table of Contents

About the Author

Rogers Cadenhead is a programmer and author. He has written more than 30 books on programming and web publishing, including *Sams Teach Yourself Java in 24 Hours* and *Absolute Beginner's Guide to Minecraft Mods Programming*. He also publishes the Drudge Retort and other websites that receive more than 20 million visits a year. He maintains this book's official website at www.java21days.com and a personal weblog at http://workbench.cadenhead.org.

Dedication

To my son Max Cadenhead, who just began his freshman year at art school.
Your talent and dedication to your craft at such an early age makes me proud.

—Dad

Acknowledgments

A book of this scope (and heft!) requires the hard work and dedication of numerous people. Most of them are at Sams Publishing in Indianapolis, and to them I owe considerable thanks—in particular, to Boris Minkin, Barbara Hacha, Elaine Wiley, and Mark Taber. Most of all, thanks to my wife, Mary, and my sons, Max, Eli, and Sam.

I'd also like to thank readers who have sent helpful comments about corrections, typos, and suggested improvements regarding this book and its prior editions. The list includes the following people: Dave Barton, Patrick Benson, Ian Burton, Lawrence Chang, Jim DeVries, Ryan Esposto, Kim Farr, Sam Fitzpatrick, Bruce Franz, Owen Gailar, Rich Getz, Bob Griesemer, Jenny Guriel, Brenda Henry-Sewell, Ben Hensley, Jon Hereng, Drew Huber, John R. Jackson, Bleu Jaegel, Natalie Kehr, Mark Lehner, Stephen Loscialpo, Brad Kaenel, Chris McGuire, Paul Niedenzu, E.J. O'Brien, Chip Pursell, Pranay Rajgarhia, Peter Riedlberger, Darrell Roberts, Luke Shulenburger, Mike Tomsic, John Walker, Joseph Walsh, Mark Weiss, P.C. Whidden, Chen Yan, Kyu Hwang Yeon, and J-F. Zurcher.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: errata@informit.com

Mail: Addison-Wesley/Prentice Hall Publishing
ATTN: Reader Feedback
330 Hudson Street
7th Floor
New York, New York, 10013

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Some revolutions catch the world by surprise. Twitter, the Linux operating system, and *Pawn Stars* all rose to prominence unexpectedly.

The remarkable success of the Java programming language, on the other hand, caught nobody by surprise. Java has been a source of great expectations since its introduction 20 years ago. When Java was introduced in web browsers, a torrent of publicity welcomed the arrival of the new language.

Sun Microsystems cofounder Bill Joy proclaimed, "This represents the end result of nearly 15 years of trying to come up with a better programming language and environment for building simpler and more reliable software."

Sun, which created Java in 1991 and first released it to the public four years later, was acquired by Oracle in 2010. Oracle, which has been committed to Java development since its earliest years, has continued to support the language and produce new versions.

In the ensuing years, Java lived up to a considerable amount of its hype. The language has become as strong a part of software development as the beverage of the same name. One kind of Java keeps programmers up nights. The other kind enables programmers to rest easier after they have developed their software.

Java was originally offered as a technology for enhancing websites with programs that run in browsers. Today, it's more likely to be found on servers, driving dynamic web applications backed by relational databases on some of the web's largest sites, and on millions of Android cell phones and tablets running popular apps such as Clash of Clans and Instagram.

Each new release of Java strengthens its capabilities as a general-purpose programming language for a wide range of environments. Today, Java is being put to use in desktop applications, Internet servers, mobile devices, and many other environments. It's even making a comeback in the browser with sophisticated applications created in Java.

Now in its ninth major release—Java 8—the Java language has matured into a full-featured competitor to other general-purpose development languages, such as C++, Python, and Ruby.

You might be familiar with Java programming tools such as Eclipse, NetBeans, and IntelliJ IDEA. These programs make it possible to develop functional Java programs, and you also can use Oracle's Java Development Kit. The kit, which is available for free on the Web, is a set of command-line tools for writing, compiling, and testing Java programs. NetBeans, another free tool offered by Oracle, is an integrated development environment for the creation of Java programs. It can be downloaded from www.netbeans.org.

This book introduces you to all aspects of Java software development using the most current version of the language and the best available techniques in Java Standard Edition, the most widely used version of the language. Programs are prepared and tested using NetBeans, so you can quickly demonstrate the skills you master each day.

Reading this book will help you understand why Java has become the most widely employed programming language on the planet.

How This Book Is Organized

Sams Teach Yourself Java in 21 Days teaches you about the Java language and how to use it to create applications for any computing environment and Android apps that run on cell phones and other mobile devices. By the time you have finished the book, you'll have well-rounded knowledge of Java and the Java class libraries. Using your new skills, you will be able to develop your own programs for tasks such as web services, database connectivity, XML processing, and mobile programming.

You learn by doing in this book, creating several programs each day that demonstrate the topics being introduced. The source code for all these programs is available on the book's official website at www.java21days.com, along with other supplemental material such as answers to reader questions.

This book covers the Java language and its class libraries in 21 days, organized into three weeks. Each week covers a broad area of developing Java programs.

In the first week, you learn about the Java language itself:

- Day 1 covers the basics—what Java is, why you should learn the language, and how to create software using a powerful style of development called object-oriented programming. You create your first Java application.
- On Day 2, you dive into the fundamental Java building blocks—data types, variables, and expressions.

- Day 3 goes into detail about how to deal with objects in Java—how to create them, use their variables, call their methods, and compare them.
- On Day 4, you give Java programs some brainpower using conditionals and work with arrays and loops.
- Day 5 fully explores creating classes—the basic building blocks of any Java program.
- On Day 6, you discover more about interfaces and packages, which are useful for grouping classes and organizing a class hierarchy.
- Day 7 covers three powerful features of Java: exceptions, the ability to deal with errors; threads, the capability to run different parts of a program simultaneously; and assertions, a technique for making programs more reliable.

Week 2 is dedicated to the most useful classes offered by Oracle for use in your own Java programs:

- Day 8 introduces data structures that you can use as an alternative to strings and arrays—array lists, stacks, maps, hash maps, and bit sets. It also describes a special `for` loop that makes them easier to use.
- Day 9 begins a five-day exploration of visual programming. You learn how to create a graphical user interface using Swing classes for interfaces, graphics, and user input.
- Day 10 covers more than a dozen interface components you can use in a Java program, including buttons, text fields, sliders, scrolling text areas, and icons.
- Day 11 explains how to make a user interface look marvelous using *layout managers*, a set of classes that determine how components on an interface are arranged.
- Day 12 concludes the coverage of Swing with event-handling classes, which enable a program to respond to mouse clicks and other user interactions.
- On Day 13, you learn about drawing shapes and characters on user interface components.
- Day 14 demonstrates how to use Java Web Start, a technique that makes installing a Java program as easy as clicking a web page link.

Week 3 moves into advanced topics:

- Day 15 covers input and output using *streams*, a set of classes that enable file access, network access, and other sophisticated data handling.

- Day 16 provides a complete introduction to closures, the most exciting new feature of Java 8. Also called *lambda expressions*, closures make it possible to employ a new type of coding called functional programming in Java for the first time. Inner classes are explored in greater depth as they relate to closures.
- On Day 17, you extend your knowledge of streams to write programs that communicate with the Internet, including socket programming, buffers, channels, and URL handling.
- Day 18 shows you how to connect to relational databases using Java Database Connectivity (JDBC) version 4.2. You learn how to exploit the capabilities of Derby, the open source database that's included with Java.
- Day 19 covers how to read and write RSS documents using the XML Object Model (XOM), an open source Java class library. RSS feeds, one of the most popular XML dialects in use today, enable millions of people to follow site updates and other new web content.
- Day 20 explores how to write web services clients with the language and the Apache XML-RPC class library.
- Day 21 covers the fastest-growing area of Java programming: developing apps for Android phones and mobile devices. Using Google's free Android Studio as a development environment and a free Android development kit, you create apps that can be deployed and tested on a phone.

Who Should Read This Book

This book teaches the Java language to three groups:

- Novices who are relatively new to programming
- People who have been introduced to earlier versions of Java
- Experienced developers in other languages, such as Visual C++, Visual Basic, or Python

When you're finished with this book, you'll be able to tackle any aspect of the Java language. You'll also be comfortable enough to tackle your own ambitious programming projects, both on and off the Web.

If you're somewhat new to programming or have never written a program, you might wonder whether this is the right book for you. Because all the concepts in this book are illustrated with working programs, you'll be able to work your way through the subject regardless of your experience level. If you understand what variables and loops are,

you'll be able to benefit from this book. You might want to read this book if any of the following are true:

- You had some beginning programming lessons in school, you grasp what programming is, and you've heard that Java is easy to learn, powerful, and cool.
- You've programmed in another language for a few years, you keep hearing accolades for Java, and you want to see whether it lives up to its hype.
- You've heard that Java is great for web application and Android programming.

If you've never been introduced to object-oriented programming, which is the style of programming that Java embodies, don't be discouraged. This book assumes that you have no background in object-oriented design. You'll get a chance to learn this development methodology as you're learning Java.

If you're a complete beginner to programming, this book might move a little fast for you. Java is a good language to start with, though, and if you take it slowly and work through all the examples, you can still pick up Java and start creating your own programs.

Conventions Used in This Book

NOTE A Note presents an interesting, sometimes technical, piece of information related to the discussion.

TIP A Tip offers advice, such as an easier way to do something.

CAUTION A Caution advises you of potential problems and helps you steer clear of disaster.

Text that you type and text that appears onscreen is presented in a `monospace` font:

```
Monospace looks like this. Hi, mom!
```

This font represents how text looks onscreen. Placeholders for variables and expressions appear in *`monospace italic`*.

The end of each lesson offers several special features: answers to commonly asked questions about that day's subject matter, a quiz to test your knowledge of the material, two exercises that you can try on your own, and a practice question in case you're preparing for Java certification. Solutions to the exercises and the answer to the certification question can be found on the book's official website at www.java21days.com.

This page intentionally left blank

DAY 3
Working with Objects

Java is an object-oriented programming language. When you do work in Java, you primarily use objects to get the job done. You create objects, modify them, change their variables, call their methods, and combine them with other objects. You develop classes, create objects out of those classes, and use them with other classes and objects.

Today, you work extensively with objects as you undertake these essential tasks:

- Creating objects
- Testing and modifying their class and instance variables
- Calling an object's methods
- Converting objects from one class to another

Creating New Objects

When you write a Java program, you define a set of classes. As you learned during Day 1, "Getting Started with Java," a class is a template used to create one or more objects. These objects, which also are called *instances*, are self-contained elements of a program with related features and data. For the most part, you use the class merely to create instances and then work with those instances. In this section, you learn how to create a new object from any given class.

When using strings during Day 2, "The ABCs of Programming," you learned that a string literal (a series of characters enclosed in double quotation marks) can be used to create a new instance of the class `String` with the value of that string.

The `String` class is unusual in that respect. Although it's a class, it can be assigned a value with a literal as if it was a primitive data type. This shortcut is available only for strings and classes that represent primitive data types, such as `Integer` and `Double`. To create instances for all other classes, the `new` operator is used.

NOTE

What about the literals for numbers and characters? Don't they create objects too? Actually, they don't. The primitive data types for numbers and characters create numbers and characters, but for efficiency they actually aren't objects. On Day 5, "Creating Classes and Methods," you learn how to use objects to represent primitive values.

Using `new`

To create a new object, you use the `new` operator with the name of the class that should be used as a template. The name of the class is followed by parentheses, as in these three examples:

```
String name = new String("Hal Jordan");

URL address = new URL("http://www.java21days.com");

MarsRobot robbie = new MarsRobot();
```

The parentheses are important and can't be omitted. They can be empty, however, in which case the most simple, basic object of that class is created. The parentheses also can contain arguments that determine the values of instance variables or other initial qualities of that object.

Here are two objects being created with arguments:

```
Random seed = new Random(606843071);

Point pt = new Point(0, 0);
```

The number and type of arguments to include inside the parentheses are defined by the class itself using a special method called a *constructor* (which is introduced later today). If you try to create a new instance of a class with the wrong number or wrong type of arguments, or if you give it no arguments and it needs them, an error occurs when the program is compiled.

Today's first project is a demonstration of creating different types of objects with different numbers and types of arguments. The `StringTokenizer` class in the `java.util` package divides a string into a series of shorter strings called *tokens*.

You divide a string into tokens by applying a character or characters as a delimiter. For example, the text "02/20/67" could be divided into three tokens—"02", "20", and "67"—using the slash character / as a delimiter.

Today's first project is a Java application that uses string tokens to analyze stock price data. In NetBeans, create a new empty Java file for the class `TokenTester` in the `com.java21days` package, and enter the code in Listing 3.1 as its source code. This program creates `StringTokenizer` objects by using `new` in two different ways and then displays each token the objects contain.

LISTING 3.1 The Full Text of `TokenTester.java`

```
 1: package com.java21days;
 2:
 3: import java.util.StringTokenizer;
 4:
 5: class TokenTester {
 6:
 7:     public static void main(String[] arguments) {
 8:         StringTokenizer st1, st2;
 9:
10:         String quote1 = "GOOG 530.80 -9.98";
11:         st1 = new StringTokenizer(quote1);
12:         System.out.println("Token 1: " + st1.nextToken());
13:         System.out.println("Token 2: " + st1.nextToken());
14:         System.out.println("Token 3: " + st1.nextToken());
15:
16:         String quote2 = "RHT@75.00@0.22";
17:         st2 = new StringTokenizer(quote2, "@");
18:         System.out.println("\nToken 1: " + st2.nextToken());
```

```
19:          System.out.println("Token 2: " + st2.nextToken());
20:          System.out.println("Token 3: " + st2.nextToken());
21:      }
22: }
```

Save this file by choosing File, Save or clicking Save All on the NetBeans toolbar. Run the application by choosing Run, Run File to see the output displayed in Figure 3.1.

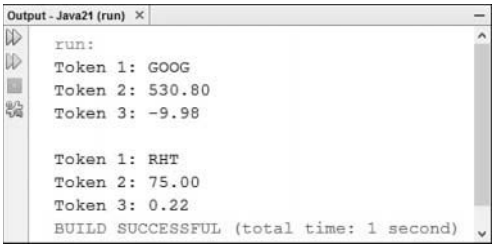


FIGURE 3.1
Displaying a `StringTokenizer` object's tokens.

Two different `StringTokenizer` objects are created using different arguments to the constructor.

The first object is created using `new StringTokenizer()` with one argument, a `String` object named `quote1` (line 11). This creates a `StringTokenizer` object that uses the default delimiters, which are blank spaces, tabs, newlines, carriage returns, or formfeed characters.

If any of these characters is contained in the string, it is used to divide the string. Because the `quote1` string contains spaces, these are used as delimiters dividing each token. Lines 12–14 display the values of all three tokens: "GOOG", "530.80", and "–9.98".

The second `StringTokenizer` object in this example has two arguments when it is constructed in line 16—a `String` object named `quote2` and an at-sign character @. This second argument indicates that the @ character should be used as the delimiter between tokens. The `StringTokenizer` object created in line 17 contains three tokens: "RHT", "75.00", and "0.22".

How Objects Are Constructed

Several things happen when you use the `new` operator. The new instance of the given class is created, memory is allocated for it, and a special method defined in the given class is called. This method is called a constructor.

A *constructor* is a way to create a new instance of a class. A constructor initializes the new object and its variables, creates any other objects that the object needs, and performs any additional operations the object requires to initialize itself.

A class can have several different constructors, each with a different number or type of argument. When you use `new`, you can specify different arguments in the argument list, and the correct constructor for those arguments is called.

In the TokenTester project, multiple constructor definitions enabled the `StringTokenizer` class to accomplish different things with different uses of the `new` operator. When you create your own classes, you can define as many constructors as you need to implement the behavior of the class.

No two constructors in a class can have the same number and type of arguments, because this is the only way constructors are differentiated from each other.

If a class defines no constructors, a constructor with no arguments is called by default when an object of the class is created. The only thing this constructor does is call the same constructor in its superclass.

CAUTION

> The default constructor only exists in a class that has not defined any constructors. Once you define at least one constructor in a class, you can't count on there being a default constructor with no arguments.

A Note on Memory Management

If you are familiar with other object-oriented programming languages, you might wonder whether the `new` operator has an opposite that destroys an object when it is no longer needed.

Memory management in Java is dynamic and automatic. When you create a new object, Java automatically allocates the proper amount of memory for that object. You don't have to allocate any memory for objects explicitly. The Java Virtual Machine (JVM) does it for you.

Because Java memory management is automatic, you don't need to deallocate the memory an object uses when you're finished using it. Under most circumstances, when you are finished with an object you have created, Java can determine that the object no longer has any live references to it. (In other words, the object isn't assigned to any variables still in use or stored in any arrays.)

As a program runs, the JVM periodically looks for unused objects and reclaims the memory that those objects are using. This process is called *dynamic garbage collection* and occurs without any programming on your part. You don't have to explicitly free the memory taken up by an object; you just have to make sure that you're not still holding onto an object you want to get rid of.

This feature is one of the most touted advantages of the language over its predecessor C++.

Using Class and Instance Variables

At this point, you can create your own object with class and instance variables, but how do you work with those variables? They're used in largely the same manner as the local variables you learned about yesterday. You can put them in expressions, assign values to them in statements, and so on. You just refer to them slightly differently.

Getting Values

To get to the value of an instance variable, you use *dot notation*, a form of addressing in which an instance or class variable name has two parts:

- A reference to an object or class on the left side of a dot operator .
- A variable on the right side

Dot notation is how you refer to an object's instance variables and methods.

For example, if you have an object named `customer` with a variable called `orderTotal`, here's how that variable could be referred to in a statement:

```
float total = customer.orderTotal;
```

This statement assigns the value of the `customer` object's `orderTotal` instance variable to a floating-point variable named `total`.

Accessing variables in dot notation is an expression (meaning that it returns a value). Both sides of the dot also are expressions. This means that you can chain instance variable access.

Extending the preceding example, suppose the `customer` object is an instance variable of the `store` class. Dot notation can be used twice, as in this statement:

```
float total = store.customer.orderTotal;
```

Dot expressions are evaluated from left to right, so you start with `store`'s instance variable `customer`, which itself has an instance variable `orderTotal`. The value of this variable is assigned to the `total` variable.

One thing to note when chaining objects together in this manner is that the statement will fail with an error if any object in the chain does not have a value yet.

Setting Values

Assigning a value to an instance variable with dot notation employs the = operator just like variables holding primitive types:

```
customer.layaway = true;
```

This example sets the value of a `boolean` instance variable named `layaway` to `true`.

The PointSetter application in Listing 3.2 tests and modifies the instance variables in a `Point` object. `Point`, a class in the `java.awt` package, represents points in a coordinate system with (x, y) values.

Create a new empty Java file in NetBeans with the class name `PointSetter` and the package name `com.java21days`; then type the source code shown in Listing 3.2 and save the file.

LISTING 3.2 The Full Text of `PointSetter.java`

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class PointSetter {
 6:
 7:     public static void main(String[] arguments) {
 8:         Point location = new Point(4, 13);
 9:
10:         System.out.println("Starting location:");
11:         System.out.println("X equals " + location.x);
12:         System.out.println("Y equals " + location.y);
13:
14:         System.out.println("\nMoving to (7, 6)");
15:         location.x = 7;
16:         location.y = 6;
17:
18:         System.out.println("\nEnding location:");
19:         System.out.println("X equals " + location.x);
20:         System.out.println("Y equals " + location.y);
21:     }
22: }
```

When you run this application, the output should match Figure 3.2.

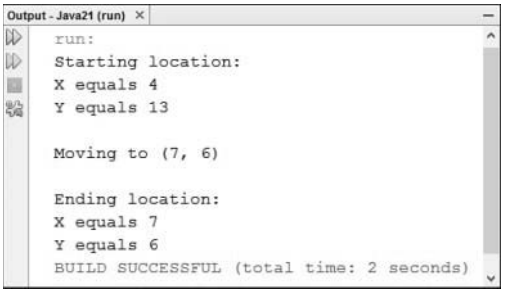


FIGURE 3.2
Setting and displaying an object's instance variables.

In this application, you create an instance of `Point` where `x` equals 4 and `y` equals 13 (line 8). These individual values are retrieved using dot notation.

The value of `x` is changed to 7 and `y` to 6 (lines 15–16). The values are displayed again to show how they have changed.

Class Variables

Class variables, as you have learned, are variables defined and stored in the class itself. Their values apply to the class and all its instances.

With instance variables, each new instance of the class gets a new copy of the instance variables that the class defines. Each instance then can change the values of those instance variables without affecting any other instances. With class variables, only one copy of that variable exists when the class is loaded. Changing the value of that variable changes it for all instances of that class.

You define class variables by including the `static` keyword before the variable itself. For example, consider the following partial class definition:

```
class FamilyMember {
    static String surname = "Mendoza";
    String name;
    int age;
}
```

Each instance of the class `FamilyMember` has its own values for `name` and `age`, but the class variable `surname` has only one value for all family members: "Mendoza." If the value of `surname` is changed, all instances of `FamilyMember` are affected.

NOTE

Calling these static variables refers to one of the meanings of the word "static": fixed in one place. If a class has a `static` variable, every object of that class has the same value for that variable.

To access class variables, you use the same dot notation as with instance variables. To retrieve or change the value of the class variable, you can use either the instance or the name of the class on the left side of the dot operator. Both lines of output in this example display the same value:

```
FamilyMember dad = new FamilyMember();
System.out.println("Family's surname is: " + dad.surname);
System.out.println("Family's surname is: " + FamilyMember.surname);
```

Because you can use an object to change the value of a class variable, it's easy to become confused about class variables and where their values are coming from. Remember that the value of a class variable affects all objects of that particular class. If the `surname` instance variable of one FamilyMember object was set to "Paciorek", all objects of that class would have that new surname.

To reduce confusion when using class variables, it's a good idea to use the name of the class when you refer to a class variable—not an object of that class. This makes the use of a class variable more clear and helps strange results become easier to debug.

Calling Methods

Methods of an object are called to make it do something.

Calling a method in an object also makes use of dot notation. The object whose method is being called is on the left side of the dot, and the name of the method and its arguments are on the right side:

```
customer.addToCart(itemNumber, price, quantity);
```

All method calls must have parentheses after them, even when the method takes no arguments, as in this example:

```
customer.cancelOrder();
```

In Listing 3.3, the StringChecker application shows an example of calling some methods defined in the `String` class. Strings include methods for string tests and modification. Create this program in NetBeans as an empty Java file with the class name `StringChecker` and package name `com.java21days`.

LISTING 3.3 The Full Text of `StringChecker.java`

```
 1: package com.java21days;
 2:
 3: class StringChecker {
 4:
 5:     public static void main(String[] arguments) {
 6:         String str = "A Lannister always pays his debts";
 7:         System.out.println("The string is: " + str);
 8:         System.out.println("Length of this string: "
 9:             + str.length());
10:         System.out.println("The character at position 6: "
11:             + str.charAt(6));
12:         System.out.println("The substring from 12 to 18: "
13:             + str.substring(12, 18));
14:         System.out.println("The index of the first 't': "
15:             + str.indexOf('t'));
16:         System.out.println("The index of the beginning of the "
17:             + "substring \"debts\": " + str.indexOf("debts"));
18:         System.out.println("The string in uppercase: "
19:             + str.toUpperCase());
20:     }
21: }
```

Running the program produces the output shown in Figure 3.3.

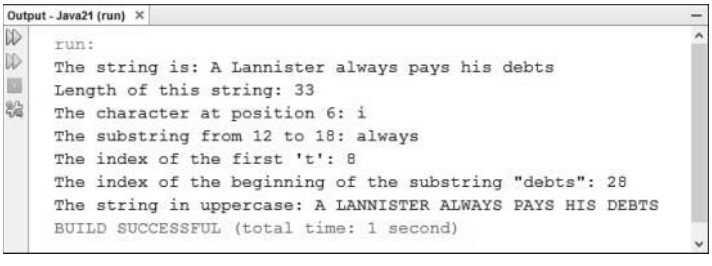


FIGURE 3.3
Calling `String` methods to learn more about that string.

In line 6, you create a new instance of `String` by using the string literal "A Lannister always pays his debts". The remainder of the program simply calls different string methods to do different operations on that string:

- Line 7 prints the value of the string.
- Line 9 calls the `length()` method in the new `String` object to find out how many characters it contains.

- Line 11 calls the `charAt()` method, which returns the character at the given position in the string. Note that string positions start at position 0 rather than 1, so the character at position 6 is 'i'.
- Line 13 calls the `substring()` method, which takes two integers indicating a range and returns the substring with those starting and ending points. The `substring()` method also can be called with only one argument, which returns the substring from that position to the end of the string.
- Line 15 calls the `indexOf()` method, which returns the position of the first instance of the given character. Character literals are surrounded by single quotation marks, so the argument is 't' (not "t").
- Line 17 shows a different use of the `indexOf()` method, which takes a string argument and returns the index of the beginning of that string. String literals always are surrounded by double quotation marks.
- Line 19 uses the `toUpperCase()` method to return a copy of the string in all uppercase.

NOTE

If you compare the output of the `StringChecker` application to the characters in the string, you might be wondering how 'i' could be at position 6 when it is the seventh character in the string. All of the methods look like they're off by one (except for `length()`). The reason is that the methods are zero-based, which means they begin counting with 0 instead of 1. So 'A' is at position 0, a space at position 1, 'L' at position 2 and so on. This kind of numbering is something you encounter often in Java.

Formatting Strings

Numbers such as money often need to be displayed in a precise manner. There are only two places after the decimal for the number of cents, a dollar sign ($) preceding the value, and commas separating groups of three numbers—as in $22,453.70 (the amount the U.S. National Debt goes up in one second).

This kind of formatting when displaying strings can be accomplished with the `System.out.format()` method.

The method takes two arguments: the output format template and the string to display. Here's an example that adds a dollar sign and commas to the display of an integer:

```
int accountBalance = 5005;
System.out.format("Balance: $%,d%n", accountBalance);
```

This code produces the output `Balance: $5,005`.

The formatting string begins with a percent sign % followed by one or more flags. The `%,d` code displays a decimal with commas dividing each group of three digits. The `%n` code displays a newline character.

The next example displays the value of pi to 11 decimal places:

```
double pi = Math.PI;
System.out.format("%.11f%n", pi);
```

The output is `3.14159265359`.

TIP

Oracle's Java site includes a beginner's tutorial for `printf`-style output that describes some of the most useful formatting codes:

http://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Nesting Method Calls

A method can return a reference to an object, a primitive data type, or no value at all. In the StringChecker application, all the methods called on the `String` object `str` returned values that are displayed. The `charAt()` method returned a character at a specified position in the string.

The value returned by a method also can be stored in a variable:

```
String label = "From";
String upper = label.toUpperCase();
```

In this example, the `String` object `upper` contains the value returned by calling `label.toUpperCase()`, which is the text "FROM".

If the method returns an object, you can call the methods of that object in the same statement. This makes it possible for you to nest methods as you would variables.

Earlier today, you saw an example of a method called with no arguments:

```
customer.cancelOrder();
```

If the `cancelOrder()` method returns an object, you can call methods of that object in the same statement:

```
customer.cancelOrder().fileComplaint();
```

This statement calls the `fileComplaint()` method, which is defined in the object returned by the `cancelOrder()` method of the `customer` object.

You can combine nested method calls and instance variable references as well. In the next example, the `putOnLayaway()` method is defined in the object stored by the `orderTotal` instance variable, which itself is part of the `customer` object:

```
customer.orderTotal.putOnLayaway(itemNumber, price, quantity);
```

This manner of nesting variables and methods is demonstrated in a method you've used frequently in the first three days of this book: `System.out.println()`.

That method displays strings and other data to the computer's standard output device.

The `System` class, part of the `java.lang` package, describes behavior specific to the computer system on which Java is running. `System.out` is a class variable that contains an instance of the class `PrintStream` representing the system's standard output, which normally is the monitor but can be a printer or file. `PrintStream` objects have a `println()` method that sends a string to that output stream. The `PrintStream` class is in the `java.io` package.

Class Methods

Class methods, also called static methods, apply to the class as a whole and not to its instances just like class variables. Class methods commonly are used for general utility methods that might not operate directly on an object of that class but do fit with that class conceptually.

For example, the `String` class contains a class method called `valueOf()`, which can take one of many types of arguments (integers, Booleans, objects, and so on). The `valueOf()` method then returns a new instance of `String` containing the argument's string value. This method doesn't operate directly on an existing instance of `String`, but getting a string from another object or data type is behavior that makes sense to define in the `String` class.

Class methods also can be useful for gathering general methods in one place. For example, the `Math` class, defined in the `java.lang` package, contains a large set of

mathematical operations as class methods. No objects can be created from the `Math` class, but you still can use its methods with numeric or Boolean arguments.

For example, the class method `Math.max()` takes two arguments and returns the larger of the two. You don't need to create a new instance of `Math`; it can be called anywhere you need it, as in the following:

```
int firstPrice = 225;
int secondPrice = 217;
int higherPrice = Math.max(firstPrice, secondPrice);
```

Dot notation is used to call a class method. As with class variables, you can use either an instance of the class or the class itself on the left side of the dot. For the same reasons noted earlier about class variables, using the name of the class makes your code easier to read.

The last two lines in this example both produce strings equal to "550":

```
String s, s2;
s = "potrzebie";
s2 = s.valueOf(550);
s2 = String.valueOf(550);
```

References to Objects

As you work with objects, it's important to understand references. A *reference* is an address that indicates where an object's variables and methods are stored.

You aren't actually using objects when you assign an object to a variable or pass an object to a method as an argument. You aren't even using copies of the objects. Instead, you're using references to those objects.

To better illustrate what this means, the RefTester application in Listing 3.4 shows how references work. Create an empty Java file in NetBeans for the class `RefTester` in the package `com.java21days`, and enter Listing 3.4 as the application's source code.

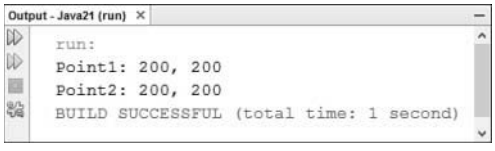
LISTING 3.4 The Full Text of `RefTester.java`

```
1: package com.java21days;
2:
3: import java.awt.Point;
4:
5: class RefTester {
6:     public static void main(String[] arguments) {
7:         Point pt1, pt2;
```

```
 8:          pt1 = new Point(100, 100);
 9:          pt2 = pt1;
10:
11:          pt1.x = 200;
12:          pt1.y = 200;
13:          System.out.println("Point1: " + pt1.x + ", " + pt1.y);
14:          System.out.println("Point2: " + pt2.x + ", " + pt2.y);
15:      }
16: }
```

Save and run the application. The output is shown in Figure 3.4.

FIGURE 3.4
Putting references to a test.



The following takes place in the first part of this program:

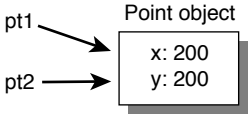
- **Line 7**—Two `Point` variables are created.
- **Line 8**—A new `Point` object is assigned to `pt1`.
- **Line 9**—The variable `pt1` is assigned to `pt2`.

Lines 11–14 are the tricky part. The `x` and `y` variables of `pt1` both are set to 200 and all variables of `pt1` and `pt2` are displayed onscreen.

You might expect `pt1` and `pt2` to have different values, but Figure 3.4 shows this not to be the case. The `x` and `y` variables of `pt2` also have changed even though nothing in the program explicitly changes them. This happens because line 7 creates a reference from `pt2` to `pt1` instead of creating `pt2` as a new object copied from `pt1`.

The variable `pt2` is a reference to the same object as `pt1`, as shown in Figure 3.5. Either variable can be used to refer to the object or to change its variables.

FIGURE 3.5
References to objects.



If you wanted `pt1` and `pt2` to refer to separate objects, you could use separate `new Point()` statements on lines 6–7 to create separate objects, as shown here:

```
pt1 = new Point(100, 100);
pt2 = new Point(100, 100);
```

References in Java become particularly important when arguments are passed to methods. You learn more about this later today.

NOTE

Java has no explicit pointers or pointer arithmetic, unlike C and C++. By using references and Java arrays, you can duplicate most pointer capabilities without many of their drawbacks.

Casting Objects and Primitive Types

One thing you discover quickly about Java is how finicky it is about the information it will handle. Like Goldilocks, the child who is oddly hard to please about porridge for a person who breaks into homes, Java methods and constructors require things to take a specific form and won't accept alternatives.

When you send arguments to methods or use variables in expressions, you must use variables of the correct data types. If a method requires an `int`, the Java compiler responds with an error if you try to send a `float` value to the method. Likewise, if you set up one variable with the value of another, they must be of compatible types. The two variables must be the same type or the variable receiving the value must be big enough to hold the value.

NOTE

There is one area where Java's compiler is decidedly flexible: the `String` object. String handling in `println()` methods, assignment statements, and method arguments is simplified by the + concatenation operator. If any variable in a group of concatenated variables is a string, Java treats the whole thing as a `String`. This makes the following possible:

```
float gpa = 2.25F;
System.out.println("Honest, mom, my GPA is a " +
(gpa + 1.5));
```

Using the concatenation operator, a single string can hold the text representation of multiple objects and primitive types in Java.

Sometimes you have a value in your Java class that isn't the right type for what you need. It might be the wrong class or the wrong data type, such as a `float` when you need an `int`.

In these situations, you can use a process called *casting* to convert a value from one type to another.

Although casting is reasonably simple, the process is complicated by the fact that Java has both primitive types (such as `int`, `float`, and `boolean`) and object types (`String`, `Point`, and the like). This section discusses three forms of casts and conversions:

- Casting between primitive types, such as `int` to `float` or `float` to `double`
- Casting from an object of a class to an object of another class, such as from `Object` to `String`
- Casting primitive types to objects and then extracting primitive values from those objects

When discussing casting, it can be easier to think in terms of sources and destinations. The source is the variable being cast into another type. The destination is the result.

Casting Primitive Types

Casting between primitive types enables you to convert the value of one type to another. This most commonly occurs with the numeric types, but one primitive type never can be used in a cast. Boolean values must be either `true` or `false` and cannot be used in a casting operation.

In many casts between primitive types, the destination can hold larger values than the source, so the value is converted easily. An example would be casting a `byte` into an `int`. Because a `byte` holds values from –128 to 127 and an `int` holds from around –2,100,000 to 2,100,000, there's more than enough room to cast a `byte` into an `int`.

Often you can automatically use a `byte` or `char` as an `int`; an `int` as a `long`; an `int` as a `float`; or anything as a `double`. In most cases, because the larger type provides more precision than the smaller, no loss of information occurs as a result. The exception is casting integers to floating-point values. Casting a `long` to a `float` or a `long` to a `double` can cause some loss of precision.

NOTE

A character can be used as an `int` because each character has a corresponding numeric code that represents its position in the character set. If the variable `key` has the value 65, the cast `(char) key` produces the character value `'A'`. The numeric code associated with a capital A is 65 in the ASCII character set, which Java adopted as part of its character support.

You must use an explicit cast to convert a value in a large type to a smaller type. Explicit casts take the following form:

```
(typename) value
```

Here *typename* is the name of the primitive data type to which you're converting, such as `short`, `int`, or `float`. *value* is an expression that results in the value of the source type. For example, in the following statement, the value of `x` is divided by the value of `y` and the result is cast into an `int`:

```
int result = (int)(x / y);
```

Note that because the precedence of casting is higher than that of arithmetic, you have to use parentheses here. Otherwise, first the value of `x` would be cast into an `int`, and then it would be divided by `y`, which could produce a different result.

Casting Objects

Objects of classes also can be cast into objects of other classes when the source and destination classes are related by inheritance and one class is a subclass of the other.

Some objects might not need to be cast explicitly. In particular, because a subclass contains all the information as its superclass, you can use an object of a subclass anywhere a superclass is expected.

For example, consider a method that takes two arguments, one of type `Object` and another of type `Component` in the `java.awt` package (which has classes for a graphical user interface).

You can pass an instance of any class for the `Object` argument because all Java classes are subclasses of `Object`.

For the `Component` argument, you can pass in its subclasses, such as `Button`, `Container`, and `Label` (all in `java.awt`).

This is true anywhere in a program, not just inside method calls. If you had a variable defined as class `Component`, you could assign objects of that class or any of its subclasses to that variable without casting.

This also is true in the reverse, so you can use a superclass when a subclass is expected. There is a catch, however: Because subclasses contain more behavior than their superclasses, a loss of precision occurs in the casting. Those superclass objects might not have all the behavior needed to act in place of a subclass object.

Consider this example: If you have an operation that calls methods in objects of the class `Integer`, using an object of its superclass `Number` won't include many methods specified in `Integer`. Errors occur if you try to call methods that the destination object doesn't have.

To use superclass objects where subclass objects are expected, you must cast them explicitly. You won't lose any information in the cast, but you gain all the methods and variables that the subclass defines.

To cast an object to another class, you use the same operation as for primitive types, which takes this form:

```
(classname) object
```

In this template, *`classname`* is the name of the destination class, and *`object`* is a reference to the source object. Casting creates a reference to the old object of the type *`classname`*; the old object continues to exist as it did before.

The following example casts an instance of the class `VicePresident` to an instance of the class `Employee`. `VicePresident` is a subclass of `Employee` with more information:

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // no cast needed for upward use
veep = (VicePresident) emp; // must cast explicitly
```

When you begin working with graphical user interfaces during Week 2, "The Java Class Library," you will see that casting one object is necessary whenever you use Java2D graphics operations. You must cast a `Graphics` object to a `Graphics2D` object before you can draw onscreen. The following example uses a `Graphics` object called `screen` to create a new `Graphics2D` object called `screen2D`:

```
Graphics2D screen2D = (Graphics2D) screen;
```

`Graphics2D` is a subclass of `Graphics` and both belong to the `java.awt` package. You'll explore this subject fully during Day 13, "Creating Java2D Graphics."

In addition to casting objects to classes, you can cast objects to interfaces, but only if an object's class or one of its superclasses actually implements the interface. Casting an object to an interface means that you can call one of that interface's methods even if that object's class does not actually implement that interface.

Converting Primitive Types to Objects and Vice Versa

One thing you can't do is cast from an object to a primitive data type, or vice versa.

Primitive types and objects are different things in Java, and you can't automatically cast between the two.

As an alternative, the `java.lang` package includes classes that correspond to each primitive data type: `Float`, `Boolean`, `Byte`, and so on. Most of these classes have the same names as the data types, except that the class names begin with a capital letter (`Short` instead of `short`, `Double` instead of `double`, and the like). Also, two classes have names that differ from the corresponding data type: `Character` is used for `char` variables and `Integer` is used for `int` variables.

Using the classes that correspond to each primitive type, you can create an object that holds the same value. The following statement creates an instance of the `Integer` class with the integer value 7801:

```
Integer dataCount = new Integer(7801);
```

After you have created an object in this manner, you can use it as you would any object (although you cannot change its value). When you want to use that value again as a primitive value, there are methods for that as well. For example, if you wanted to get an `int` value from a `dataCount` object, the following statement shows how:

```
int newCount = dataCount.intValue(); // returns 7801
```

A common translation you need in programs is converting a `String` to a numeric type, such as an integer. When you need an `int` as the result, this can be done by using the `parseInt()` class method of the `Integer` class. The `String` to convert is the only argument sent to the method, as in the following example:

```
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

The following classes can be used to work with objects instead of primitive data types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, and `Void`. These classes are commonly called *object wrappers* because they provide an object representation that contains a primitive value.

CAUTION

If you try to use the preceding example in a program, your program won't compile. The `parseInt()` method is designed to fail with a NumberFormatException error if the argument to the method is not a valid numeric value. To deal with errors of this kind, you must use special error-handling statements, which are introduced during Day 7, "Exceptions and Threads."

Working with primitive types and objects that represent the same values is made easier through autoboxing and unboxing, an automatic conversion process.

Autoboxing automatically converts a primitive type to an object. *Unboxing* converts in the other direction.

If you write a statement that uses an object where a primitive type is expected, or vice versa, the value is converted so that the statement executes successfully.

Here's an example of autoboxing and unboxing:

```
Float f1 = 12.5F;
Float f2 = 27.2F;
System.out.println("Lower number: " + Math.min(f1, f2));
```

The `Math.min()` method takes two `float` values as arguments, but the preceding example sends the method two `Float` objects as arguments instead.

The compiler does not report an error over this discrepancy. Instead, the `Float` objects automatically are unboxed into `float` values before being sent to the `min()` method.

CAUTION

Unboxing an object works only if the object has a value. If no constructor has been called to set up the object, compilation fails with an error.

Comparing Object Values and Classes

In addition to casting, you often will perform three other common tasks that involve objects:

- Comparing objects
- Finding out the class of any given object
- Testing whether an object is an instance of a given class

Comparing Objects

Yesterday, you learned about operators for comparing values—equal, not equal, less than, and so on. Most of these operators work only on primitive types, not on objects. If you try to use other values as operands, the Java compiler produces errors.

The exceptions to this rule are the == operator for equality and the != operator for inequality. When applied to objects, these operators don't do what you might first expect. Instead of checking whether one object has the same value as the other, they determine whether both sides of the operator refer to the same object.

To compare objects of a class and have meaningful results, you must implement special methods in your class and call those methods.

A good example of this is the `String` class. It is possible to have two different `String` objects that represent the same text. If you were to employ the == operator to compare these objects, however, they would be considered unequal. Although their contents match, they are not the same object.

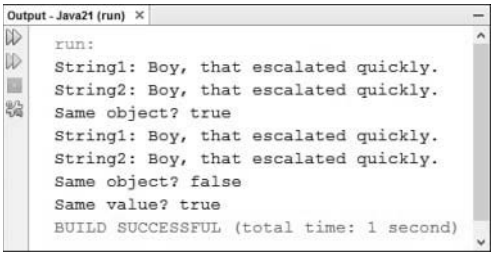
To see whether two `String` objects have matching values, an `equals()` method of the class is used. The method tests each character in the string and returns `true` if the two strings have the same value. The EqualsTester application shown in Listing 3.5 illustrates this. Create the application with NetBeans in the `com.java21days` package and save the file, either by choosing File, Save or clicking the Save All toolbar button.

LISTING 3.5 The Full Text of `EqualsTester.java`

```
 1: package com.java21days;
 2:
 3: class EqualsTester {
 4:     public static void main(String[] arguments) {
 5:         String str1, str2;
 6:         str1 = "Boy, that escalated quickly.";
 7:         str2 = str1;
 8:
 9:         System.out.println("String1: " + str1);
10:         System.out.println("String2: " + str2);
11:         System.out.println("Same object? " + (str1 == str2));
12:
13:         str2 = new String(str1);
14:
15:         System.out.println("String1: " + str1);
16:         System.out.println("String2: " + str2);
17:         System.out.println("Same object? " + (str1 == str2));
18:         System.out.println("Same value? " + str1.equals(str2));
19:     }
20: }
```

The program's output appears in Figure 3.6.

FIGURE 3.6
Calling `String` methods to learn more about that string.



The first part of this program declares two variables (`str1` and `str2`), assigns the literal "Boy, that escalated quickly." to `str1`, and then assigns that value to `str2` (lines 5–7). As you learned earlier, `str1` and `str2` now point to the same object, and the equality test at line 11 proves that.

In the second part of this program, you create a new `String` object with the same value as `str1` and assign `str2` to that new `String` object.

Now you have two different string objects in `str1` and `str2`, both with the same value. Testing them to see whether they're the same object by using the == operator returns the expected answer: `false` (line 17). They are not the same object in memory. Testing them using the `equals()` method in line 18 also returns the expected answer of `true`, which shows they have the same value.

NOTE

Why can't you just use another literal when you change `str2`, instead of using `new`? String literals are optimized in Java. If you create a string using a literal and then use another literal with the same characters, Java gives you back the first `String` object. Both strings are the same object; you have to go out of your way to create two separate objects.

Determining the Class of an Object

Want to find out the name of an object's class? Here's how for an object assigned to the variable `key`:

```
String name = key.getClass().getName();
```

The `getClass()` method is defined in the `Object` class, so it can be called in all Java objects. The method returns a `Class` object that represents the object's class. That object's `getName()` method returns a string holding the name of the class.

Another useful test is the `instanceof` operator, which has two operands: a reference to an object on the left, and a class name on the right. The expression produces a Boolean value: `true` if the object is an instance of the named class or any of that class's subclasses, or `false` otherwise, as in these examples:

```
boolean check1 = "Texas" instanceof String; // true

Object obiwan = new Object();
boolean check2 = obiwan instanceof String; // false
```

The `instanceof` operator also can be used for interfaces. If an object implements an interface, the `instanceof` operator returns `true` when this is tested.

Unlike other operators in Java, `instanceof` is not a form of punctuation like * for multiplication or + for addition. Instead, the `instanceof` keyword is the operator.

Summary

Now that you have spent three days exploring how object-oriented programming is implemented in Java, you're in a better position to decide how useful it can be in your programming.

If you are a "glass half empty" kind of person, object-oriented programming (OOP) is a level of abstraction that gets in the way of using a programming language. You learn more about why OOP is thoroughly ingrained in Java in the coming days and may change your mind.

If you are a "glass half full" kind of person, object-oriented programming is beneficial because of its benefits: improved reliability, reusability, and maintenance.

Today you learned how to deal with objects: creating them, reading their values and changing them, and calling their methods. You also learned how to cast objects from one class to another, cast to and from primitive data types and classes, and take advantage of automatic conversions through autoboxing and unboxing.

Q&A

Q I'm confused about the differences between objects and the primitive data types, such as `int` and `boolean`.

A The primitive types (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`) are not objects, although in many ways they can be handled like objects. They can be assigned to variables and passed in and out of methods.

Objects are instances of classes and as such are much more complex data types than simple numbers and characters. They often contain numbers and characters as instance or class variables.

Q The `length()` and `charAt()` methods in the StringChecker application (Listing 3.3) don't appear to make sense. If `length()` says that a string is 33 characters long, shouldn't the characters be numbered from 1 to 33 when `charAt()` is used to display characters in the string?

A The two methods look at strings differently. The `length()` method counts the characters in the string, with the first character counting as 1, the second as 2, and so on. The `charAt()` method considers the first character in the string to be located at position number 0. This is the same numbering system used with array elements in Java. Consider the string "Charlie Brown". It has 13 characters ranging from position 0 (the letter C) to position 12 (the letter n).

Q If Java lacks pointers, how can I do something like linked lists, where there's a pointer from one node to another so that they can be traversed?

A It's incorrect to say that Java has no pointers; it just has no *explicit* pointers. Object references are effectively pointers. To create something like a linked list, you could create a class called `Node`, which would have an instance variable also of type `Node`. To link node objects, assign a node object to the instance variable of the object immediately before it in the list. Because object references are pointers, linked lists set up this way behave as you would expect them to. (You'll work with the Java class library's version of linked lists on Day 8, "Data Structures.")

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What operator do you use to call an object's constructor and create a new object?
   - A. `+`
   - B. `new`
   - C. `instanceof`
2. What kind of methods apply to all objects of a class rather than an individual object?
   - A. Universal methods
   - B. Instance methods
   - C. Class methods
3. If you have a program with objects named `obj1` and `obj2`, what happens when you use the statement `obj2 = obj1`?
   - A. The instance variables in `obj2` are given the same values as `obj1`.
   - B. `obj2` and `obj1` are considered to be the same object.
   - C. Neither A nor B.

Answers

1. B. The `new` operator is followed by a call to the object's constructor.
2. C. Class methods can be called without creating an object of that class.
3. B. The = operator does not copy values from one object to another. Instead, it makes both variables refer to the same object.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

```
public class AyeAye {
    int i = 40;
    int j;
```

```
    public AyeAye() {
        setValue(i++);
    }

    void setValue(int inputValue) {
        int i = 20;
        j = i + 1;
        System.out.println("j = " + j);
    }
}
```

What is the value of the `j` variable at the time it is displayed inside the `setValue()` method?

A. 42

B. 40

C. 21

D. 20

The answer is available on the book's website at www.java21days.com. Visit the Day 3 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a program that turns a birthday in MM/DD/YYYY format (such as 04/29/2016) into three individual strings.
2. Create a class with instance variables for `height`, `weight`, and `depth`, making each an integer. Create a Java application that uses your new class, sets each of these values in an object, and displays the values.

Exercise solutions are offered on the book's website at www.java21days.com.

This page intentionally left blank

This page intentionally left blank

Index

Symbols

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

W

X–Z