JAVA
SUCCINCTLY
PART 1

BY **CHRISTOPHER ROSE**

Java Succinctly Part 1

By

Christopher Rose

Foreword by Daniel Jebaraj

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.
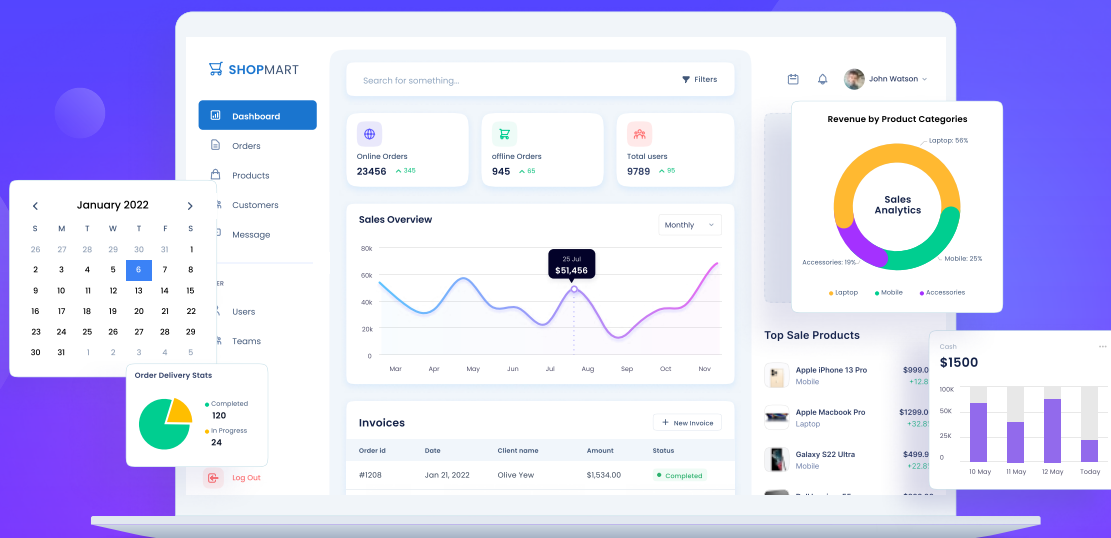
Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Christopher Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Majestic Theatre in Pomona, Queensland.

Chapter 1 Introduction

This e-book provides an introduction to programming in Java. It covers the fundamental building blocks of the language and is designed for readers who have little or no programming background. A second e-book, *Java Succinctly Part 2*, will cover the language in more advanced terms, with topics such as GUI and multithreading. Code samples can be found [here](.).

Java is a high-level, cross-platform, object-oriented programming language. The main objective of Java is to allow applications to be written once and run on a multitude of different devices. Java achieves its cross-platform nature by using a virtual machine architecture called the Java Virtual Machine (JVM). Java applications are ubiquitous, and the language is consistently ranked as one of the most popular and dominant in the world. It can be found everywhere from mobile phones to web applications to full desktop applications.

Java is a curly bracket, or curly brace, language. This means it uses { and } braces to designate blocks of code. The language inherits its roots from older languages such as C and C++. The curly bracket languages share many similarities, and once a programmer is familiar with one of them, it is not difficult to pick up another. Out of hundreds of computer programming languages used all over the world, Java is possibly the best introduction to programming because it offers a degree of safety to new programmers while maintaining the power and flexibility of its low-level ancestry.

Java is an object-oriented programming language. Object-oriented programming is a very important programming paradigm that allows the programmer to create virtual modules of code for reuse and to separate a program into components. There are many other object-oriented programming languages, and Java offers the perfect introduction to how the paradigm operates.

The main objective of this e-book is to describe the foundations of Java—from printing a line of text to the console all the way up to inheritance hierarchies in object-oriented programming. We will cover practical aspects of programming, such as debugging and using an IDE, as well as the core mechanics of the language.

If you are new to programming computers, or if you simply want to brush up on some core Java, I hope you have fun with the various challenges offered here.

Introduction to computer programming

Computers are machines that read and write data to various areas of memory. Computers can seem very complicated when we see a video game or complex statistics software. But in reality, computers are extremely simple. They are capable of only two things: reading and writing numbers. When we program a computer, we specify exactly what the computer should read and what the computer should write.
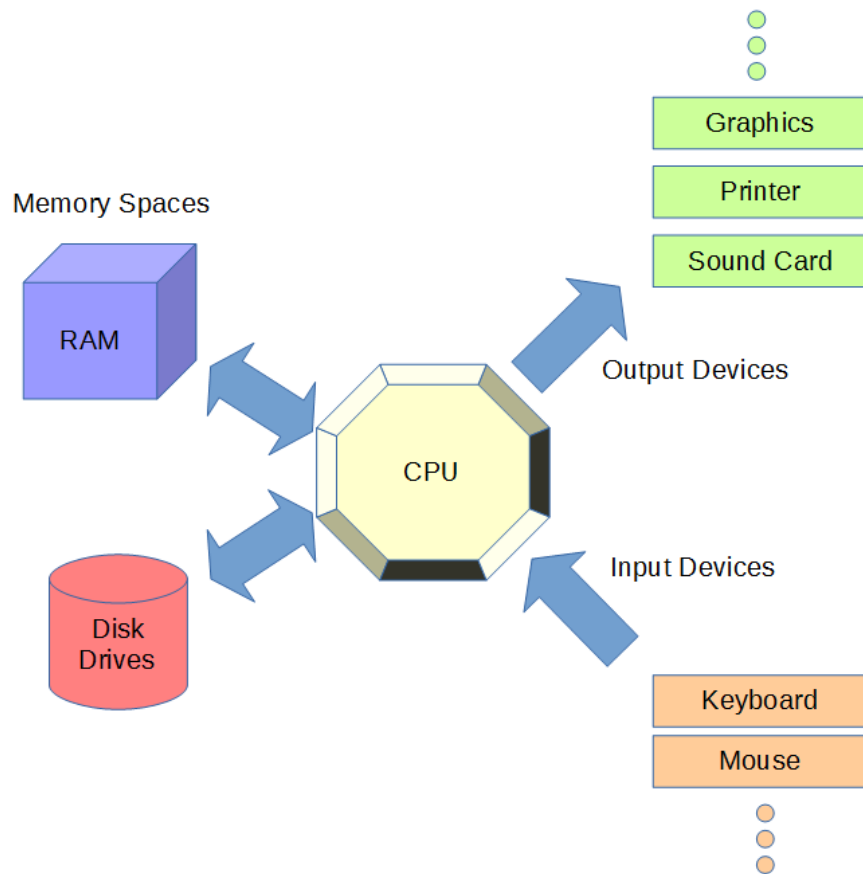
Figure 1: Outline of a Computer

Figure 1 illustrates the basic features of a computer. In the center, we have a very important piece of hardware, the Central Processing Unit (CPU), which is the brains of the machine. It reads input as a string of numbers from the keyboard, mouse, or other input device. It writes data to the graphics, printers, and sound devices. It also reads and writes data to memory spaces such as RAM and hard drives.

A CPU operates by executing a string of instructions. Everything in a computer is a number, even the instructions themselves. The instructions are generally very simple—they indicate that the CPU should perform a task such as adding two numbers or creating a copy of a number. The language of the CPU—Assembly language—is the lowest level of computer programming. Programming a CPU in its Assembly language is cumbersome because each instruction does almost nothing, and we need many thousands of instructions before our applications are useful. And every CPU has its own Assembly language. If we program an application to run on desktop PCs, the code will not be understood by an iPhone or a Samsung tablet.

Because it is difficult to program a CPU in its own language, people have invented high-level languages that are much closer to human languages (such as English and arithmetic). Java is one such high-level language. Instead of specifying a string of individual instructions, we can use mathematical expressions like a = a + b. This saves us a lot of time for two reasons:

- It is easier to program.
- It is easier to read and maintain.

```
; Assembly Language: Low Level Language

mov rax, a ; Read the value of a

add rax, b ; Add the value of b to a

mov a, rax ; Write the result to a



// Java: High Level Language

a = a + b;
```

Code Listing 1.0 shows a clear distinction between a high-level language (such as Java) and a low-level language (such as Assembly Language). In the low-level language, the addition of two values, `a` and `b`, and the storage of the result back into `a` takes three steps. The computer must be instructed to perform every specific detail of every operation, from moving the values into internal registers, such as `RAX`, to the addition itself. In Java, we can express this entire operation as a single, easy-to-read line of code that looks exactly like middle school algebra: a = a + b. The line means: Take the value in the variable a, add to it the value of b, and assign the result back to the variable a.

Code Listing 1.0 shows a simple expression in Java, and this expression would look the same in many other C-based languages. However, there is an additional advantage to programming an application in Java as opposed to C or C++. Java programs are designed to run on a virtual machine. Figure 2 shows an illustration of native applications versus a Java application. When we write native applications, we must compile our code for each type of system on which we need our application to run. This means we will need to create a version for ARM CPUs if we want our application to run on mobile devices and also create another version of the programming for x86 if we want our application to run on desktops and laptops.

Java does not work like C and C++ in this respect. In Java, we program a single version of our application. The different CPUs (ARM, x86, etc.) all run the Java Virtual Machine, which, in turn, runs our application. The Virtual Machine is simply a program written to emulate the hardware of the hypothetical Java computer. This means we can write our program once, and any device that has a Java Virtual Machine emulator (i.e. has Java installed on it) will be capable of running our applications. This is Java's massive advantage—it means our applications will run on a very large number of devices without needing us to design specifically for them.
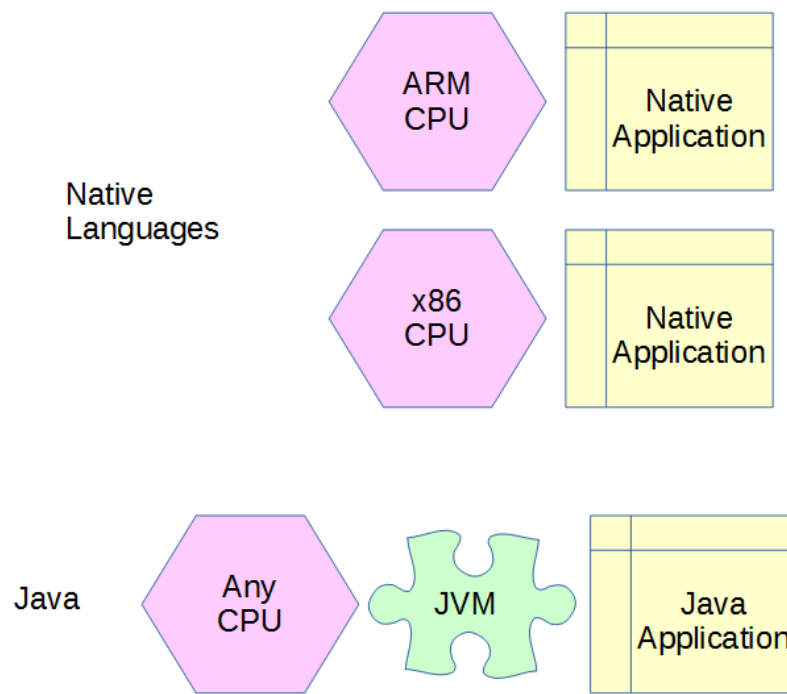
Figure 2: Java Virtual Machine

Chapter 2 Getting Started

In this section, we will examine how to install the Eclipse Integrated Development Environment (IDE). Eclipse is a tool used to program, maintain, debug, and compile Java applications. An IDE is a collection of tools that make application development easier. Java, like many languages, can be programmed with nothing more than a console and a text editor (such as Notepad). However, using Notepad as an IDE is laborious and can lead to errors. A fully functional IDE such as Eclipse can greatly reduce the amount of time it takes to solve problems in a language. Eclipse offers syntax highlighting, bug-fixing suggestions, single-click testing and debugging of an application, and literally thousands of other options for helping to make programming simpler.

Downloading and installing Eclipse

To get the latest version of Eclipse, head over to https://eclipse.org/downloads/.
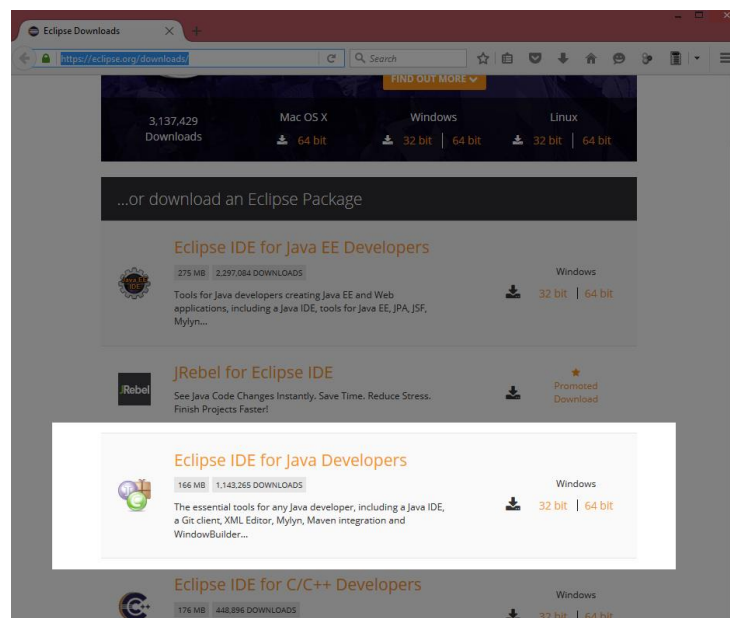
Figure 3: Eclipse Download Page

There are several versions of Eclipse available for download (and several languages for which Eclipse can be used to develop). I have used the Eclipse IDE for Java Developers in this e-book, the version is Mars.1 Release (4.5.1). Older and newer versions of Eclipse will also be fine. At the time of writing, the top of the main Eclipse download page also offers the Eclipse installer, which is an installation wizard designed to make installing Eclipse simple. If you have trouble installing Eclipse with the manual method described here, you might try installing with the installation wizard.

On the right side of the highlighted region in Figure 4, you have two options: 32-bit and 64-bit. I recommend that you use the 64-bit version of Eclipse unless you are certain that your machine is only 32 bits.

When you click the orange button marked "64-bit" (or the button marked "32-bit"), you will be taken to the site for selecting a mirror. This is the source from which to download Eclipse. Choose the mirror that is closest to your region (this is most likely the default mirror displayed on the left). See Figure 4.
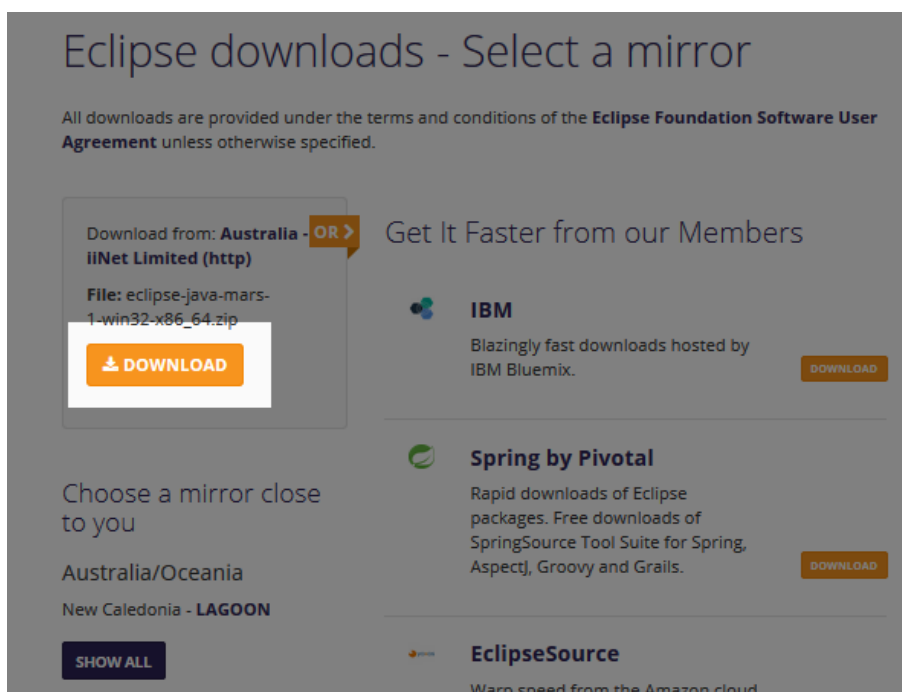

Figure 4: Selecting a Mirror

Selecting different mirrors may alter the amount of time needed for your download to complete, and, generally speaking, the nearer the mirror to your geographical location, the faster you can download Eclipse.

After you select the mirror, you will be taken to a page that allows you to donate to the Eclipse foundation if you wish. The download should begin. You will be asked where you want the file to download. The exact mechanisms for saving files are different depending on your web browser, but make sure that you know where the download is going because we need to extract the file when it is finished downloading. When the download is complete, you will have a file called something similar to "*eclipse-java-mars-1-win32-x86_64.zip,*" which is a compressed Zip archive of the Eclipse program.

When the download has finished, find this file and extract it. The exact location where you wish to place Eclipse is entirely up to you, but for the 32-bit version, it is probably a good idea to place it in an application folder such as "C:\Program Files" or "C:\Program Files (x86)." Create the folder you want to place Eclipse into, copy the Zip file into this folder, and extract it by right-clicking the file and selecting the **Extract to** option (as per Figures 5 and 6).
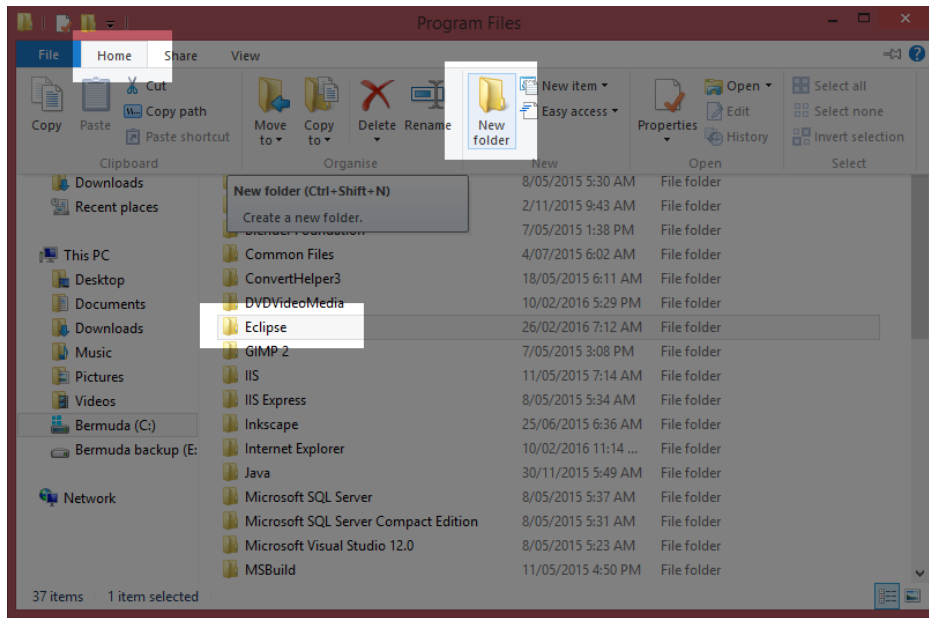
Figure 5: Creating the Eclipse Folder
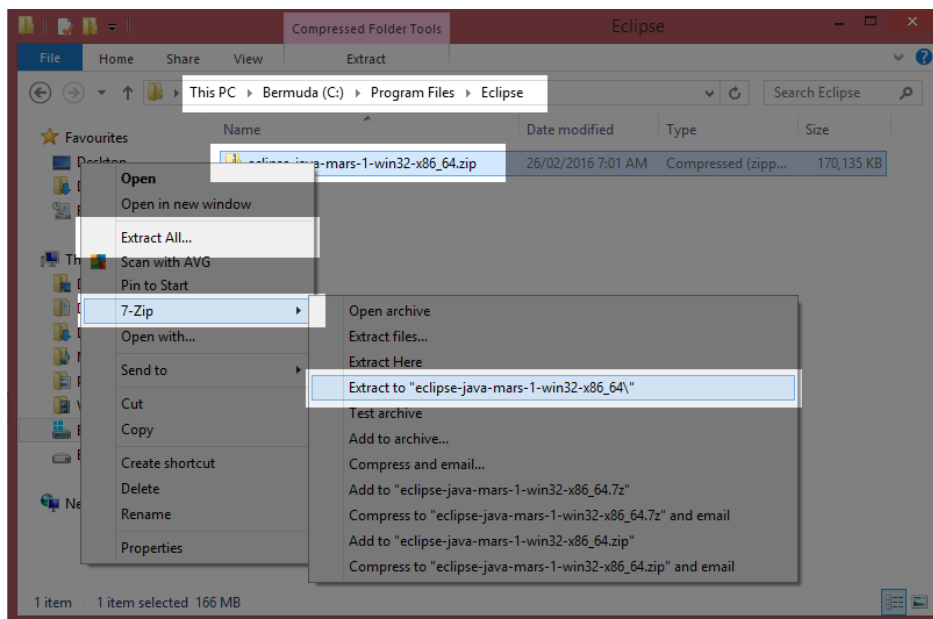
Figure 6: Extracting the Eclipse Zip File

Note: On Windows 8 and above, we are able to extract Zip files by right-clicking the file and selecting "Extract All" from the context menu. Alternately, you can install dedicated archiving extraction and compression software. I recommend the program 7-Zip, which is a powerful and flexible freeware-compression tool.

After we use "Extract All" or some dedicated extraction tool to extract the contents of the Eclipse Zip file, we will have a subfolder in the Eclipse folder called "eclipse-java-***" in which the *** is the version of Eclipse. Navigate into this folder and into the following "eclipse" subfolder, and

you will see the main Eclipse files, as per Figure 7. In order to start the Eclipse IDE, double-click the file called **Eclipse.exe**.

Figure 7: Extracted Eclipse Files

Tip: If you right-click the file Eclipse.exe, you can select the option "Create Shortcut." Clicking this option will create a shortcut to the Eclipse.exe file in the same folder as the EXE. You can then cut and paste this shortcut to your desktop for easy access.

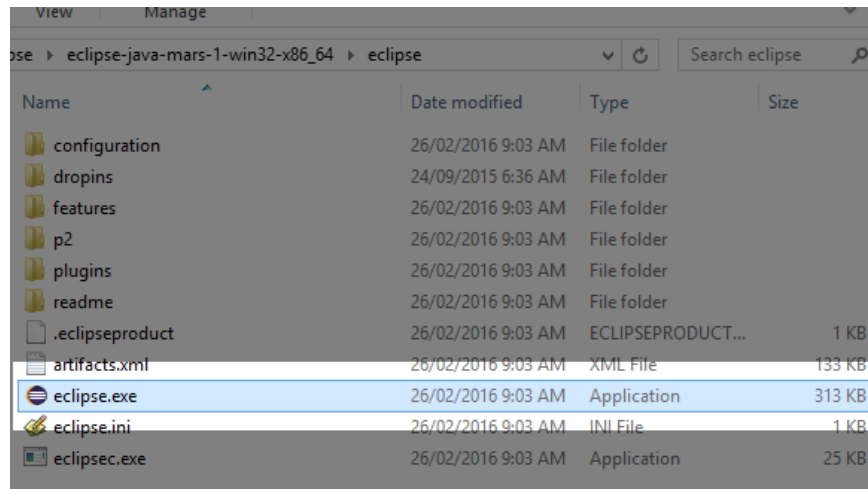*Note: You will also need the Java Runtime Environment installed (this is installed automatically if you selected to download the Eclipse installer package from the website above). If you do not have the Java Runtime Environment installed, you can download and install it from http://www.oracle.com/technetwork/java/javase/downloads/index.html.*

Projects and workspaces

A Java project is a collection of classes, files, and other resources that collectively form a coherent application. A workspace is a collection of Java projects in a folder on the developer's machine. In this section, we will look at creating a new Java project in Eclipse, adding a main class and an application entry point. Our program will not do anything visible at this point.

Workspaces

Eclipse organizes projects into folders called Workspaces. You can create a new workspace for each project, or you can use a single workspace for several related projects, such as libraries and a front end that uses them. When you first run Eclipse, you will be asked to specify the workspace in which you would like to store your applications while you develop. You can create more workspaces later, but for now let's use the default workspace—located in the user's folder (*C:\Users\Chris* in my case) and in a subfolder called "workspace" (see Figure 8).
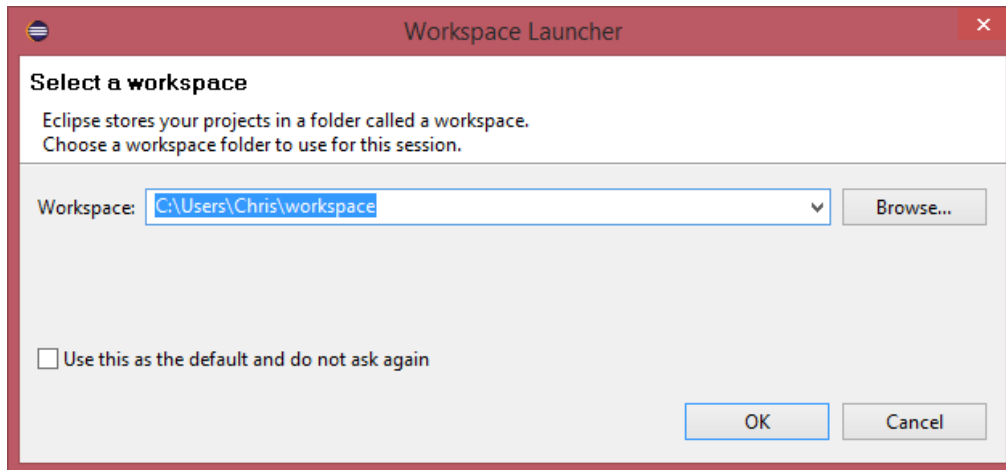
Figure 8: Selecting a Workspace

Selecting the option "Use this as the default and do not ask again" will cause all subsequent projects you create to be created in the same workspace. This is a good idea, except that eventually you might have hundreds of projects and locating the projects you need to work on can become tedious.

Projects

In order to program a Java application, we need to create a Java project. A project is simply a collection of related files, folders, and resources that form an application. In order to begin a new project, click **New** in the File menu, then **Java Project**, as per Figure 9.
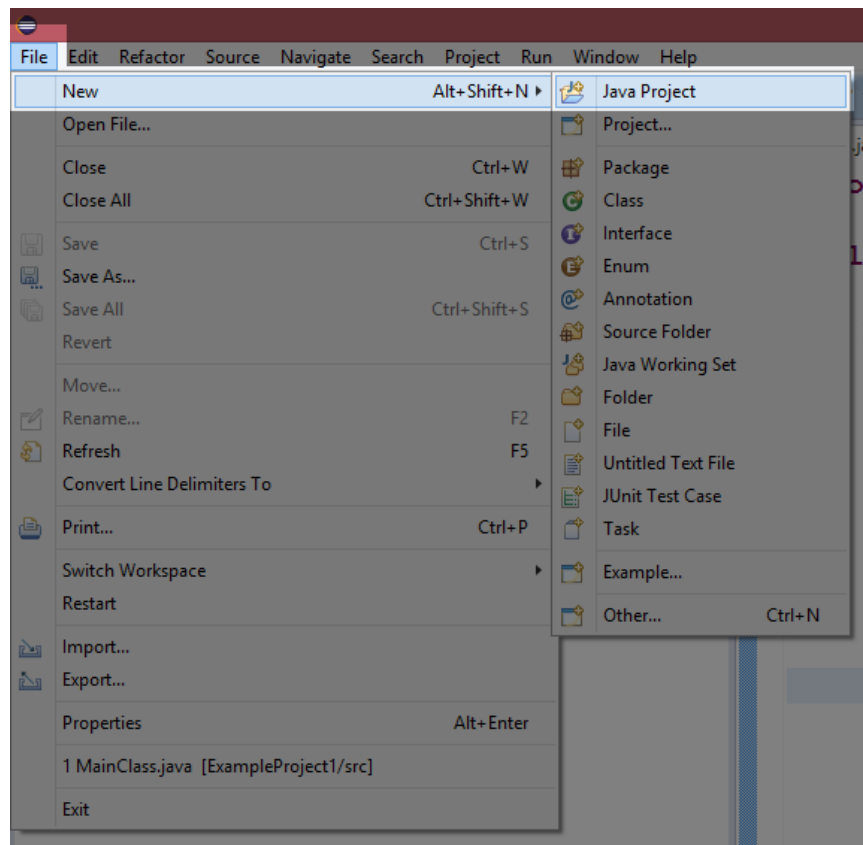
Figure 9: Creating a New Project

When you create a new project, Eclipse will show the New Project Wizard (Figure 10).
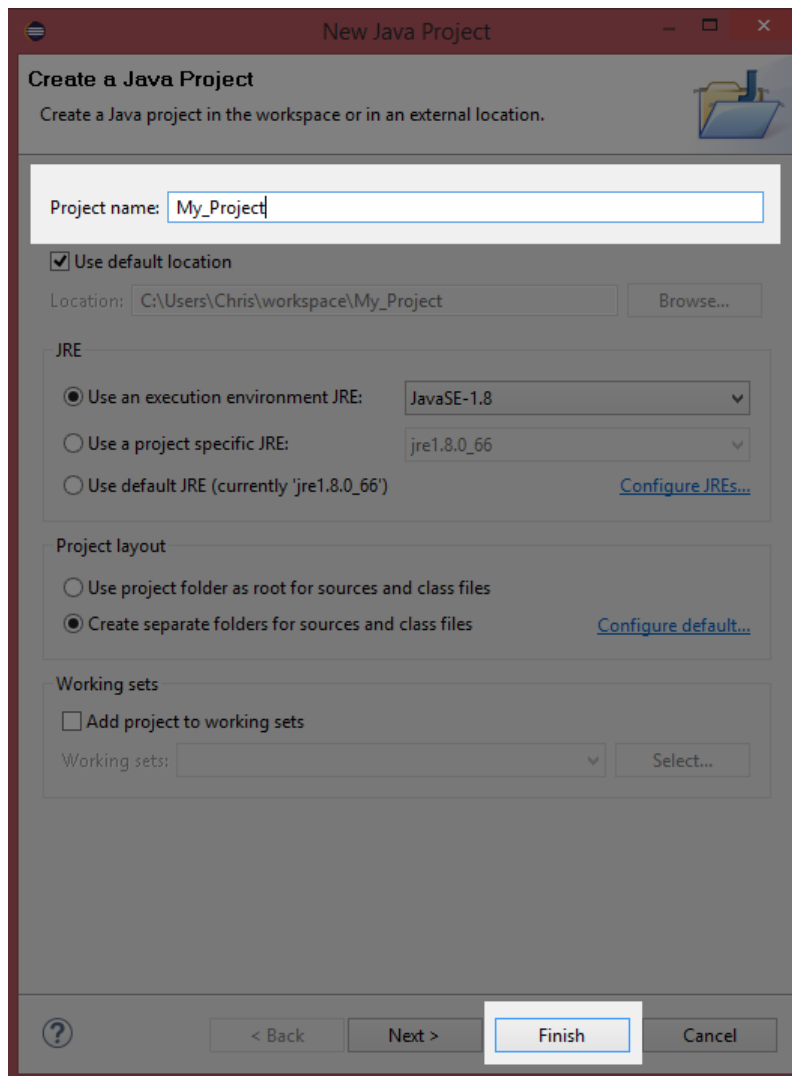
Figure 10: New Java Project Wizard

In the New Java Project Wizard dialog box, give your project a name and click **Finish** (the name of each project must be unique within the workspace—I have called the project My_Project in Figure 10). If you select Next instead of clicking Finish, you can change several more advanced features of your project, such as the output folder, and whether or not you wish to reference other existing projects.

Package explorer

After you have selected Finish in the New Java Project Wizard, Eclipse will create a basic project for you, including an src folder for storing your Java source files and a collection of system libraries that are referenced by default. The JRE System Libraries have a lot of useful functions that keep us from having to program them ourselves. We will examine these later.

The package explorer (Figure 11) contains a tree view of all the projects in the current workspace. At the moment, it only holds a single project called My_Project. The nodes of the tree view can be expanded and collapsed by clicking the small triangles on the left.
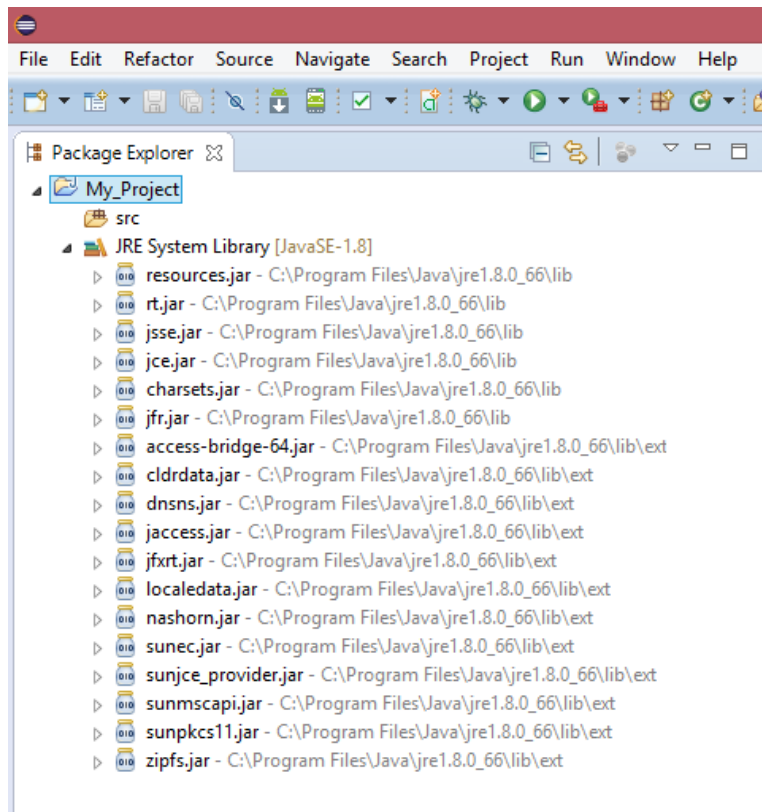
Figure 11: Package Explorer

Java source code

Source code is a term meaning the text of a particular computer programming language. Java source is typed into plain text files that we add to our projects as classes. The Java Compiler (a program called *javac.exe* that was installed with the Java SDK along with Eclipse), reads the source code files in a project and converts them into machine code. The machine code can be understood and executed as an application by the JVM.

Before we do anything else, we want to add a new Java class to our project. We will explore the meaning of the term "class" when we look at object-oriented programming. For now, our class will give us a starting point for our application. In the package explorer, right-click the **src** folder and select **New** and **Class** from the context menus that open, as per Figure 12. The src folder is the main folder for storing our source code files.

Figure 12: Adding a New Class
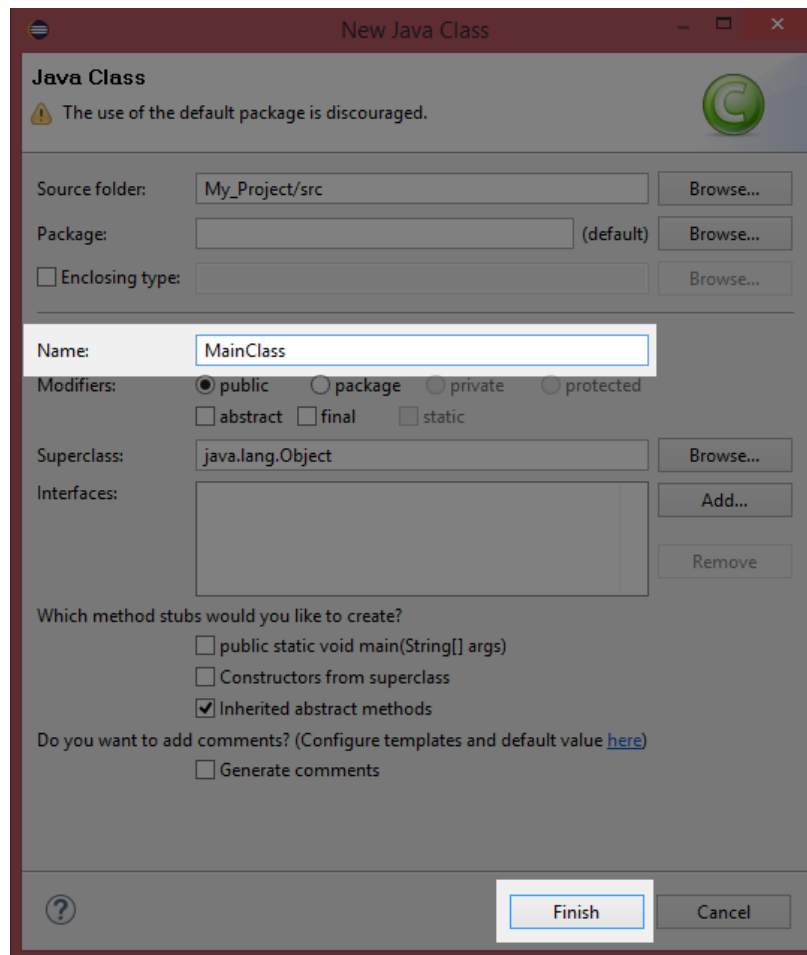
Figure 13: New Java Class Wizard

In the New Java Class window, type a name for your new class, such as MainClass, then click **Finish** in order to continue. Eclipse will create a new source code file for you and will fill out the basic skeleton of a Java class.
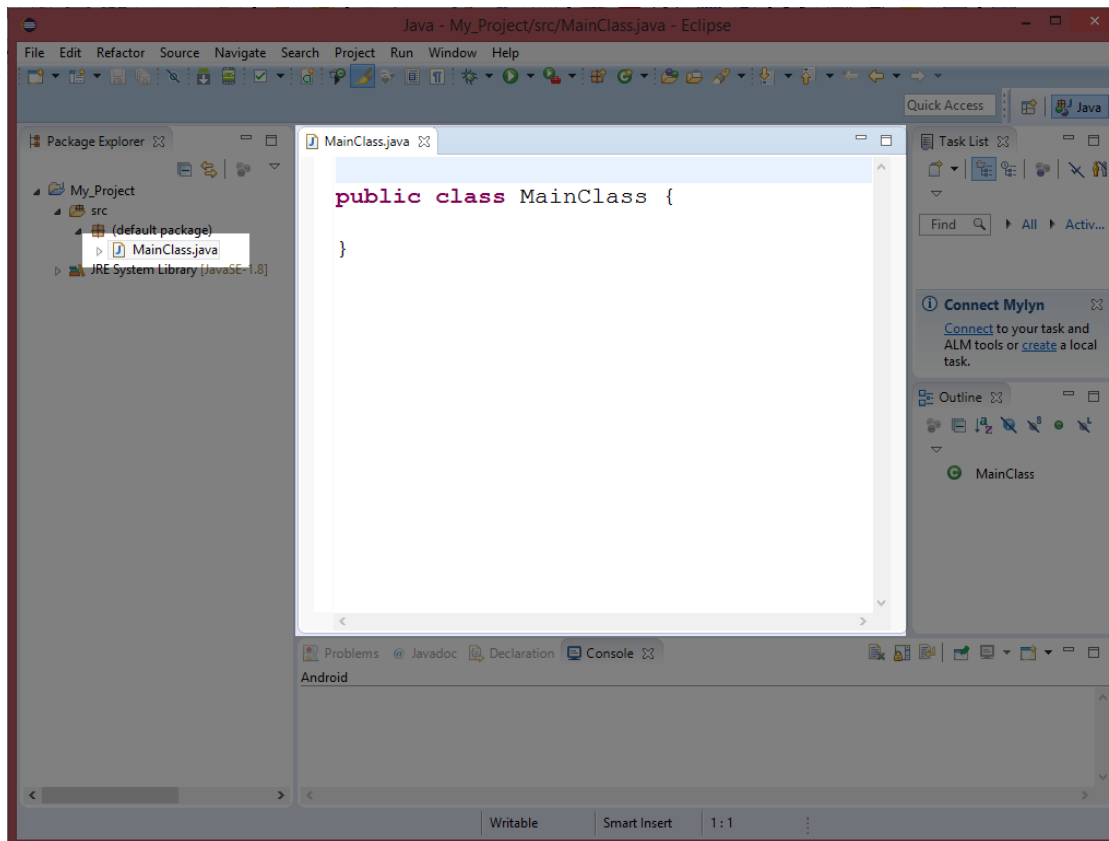
Figure 14: The Code Window

In Figure 14, the code window shows the beginnings of our first program. We type our Java source code into this window. At the moment, it has the dec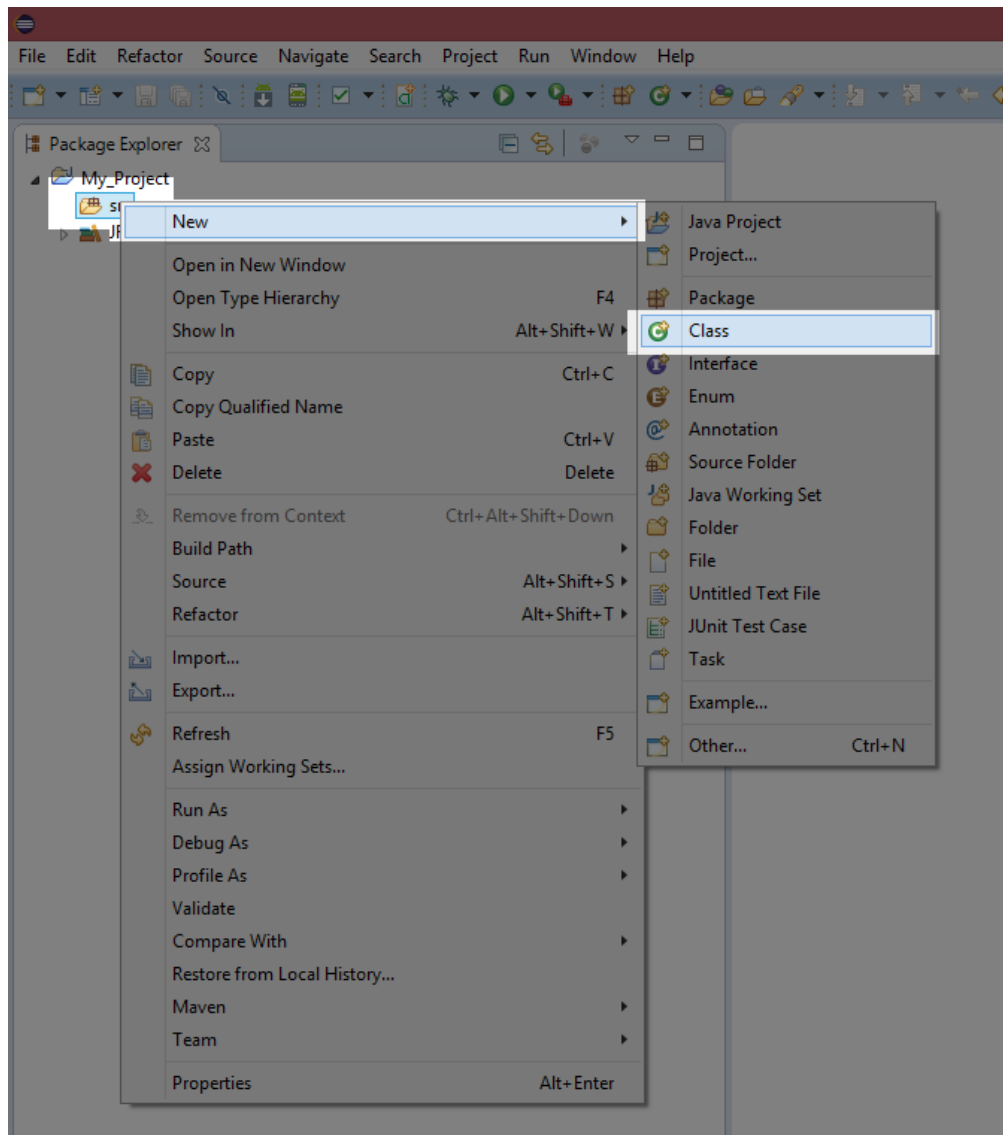laration of the class we requested, but nothing else. If you cannot see the MainClass.java code in your source code window, try locating the file in the package explorer—expand the src node and double-click the file called **MainClass.java**.

We will add many classes to our projects in future chapters. The method for adding a class is always the same—right-click the **src** folder in the package explorer, select **New** and **Java class**, and give the class a unique name.

Do-nothing program

At the moment, we have a project, but it has no entry point. There is no place for the JVM to begin executing our code. The default entry point for a Java application is called the `main` method. It consists of a `public`, `static` method called `main` that takes an array of strings as parameters. We will look at exactly what this means later. For now, in order to include a `main` method we need to type the declaration of the method and add braces for the body inside the `MainClass` (see Code Listing 2.0).

```java
public class MainClass {
public static void main(String[] args) {
      }
}
```

Code Listing 2.0 shows the simplest Java application possible. It is an application that does nothing but exit the moment it is run. When the JVM is asked to load an application (either by debugging our app or running it), it looks for an entry point. The default entry point is a method called `main` with the exact definition as per Code Listing 2.0: **`public static void` `main(String[] args)`**.

Note: Java is case sensitive. If you choose to type the code in the example listings in this e-book rather than copying and pasting the code, you must be very careful to match case exactly. This means that the compiler sees the terms PUBLIC, Public, PuBllc, and public as completely different terms.

If there is no method exactly matching the definition of the `main` method in Code Listing 2.0, the JVM will not know where to start executing code and the program will not run.

Note: The main method must have the exact declaration, except for the identifier name of the args parameter. This parameter is simply an array of strings that can potentially be passed to our application when invoking from the command line. When we invoke a program from the command line, we can optionally supply a list of arguments (hence the name args for this parameter). For our purposes, we will ignore this parameter and assume it will always be empty, and we will always call it args.

Project output

Eclipse stores all the files related to our project in the folder we specify at start up. Figure 15 shows the folder in Windows Explorer. You will notice that there is an src folder that was visible in the package explorer. When we add source files to our package, they will in turn be added to this folder.
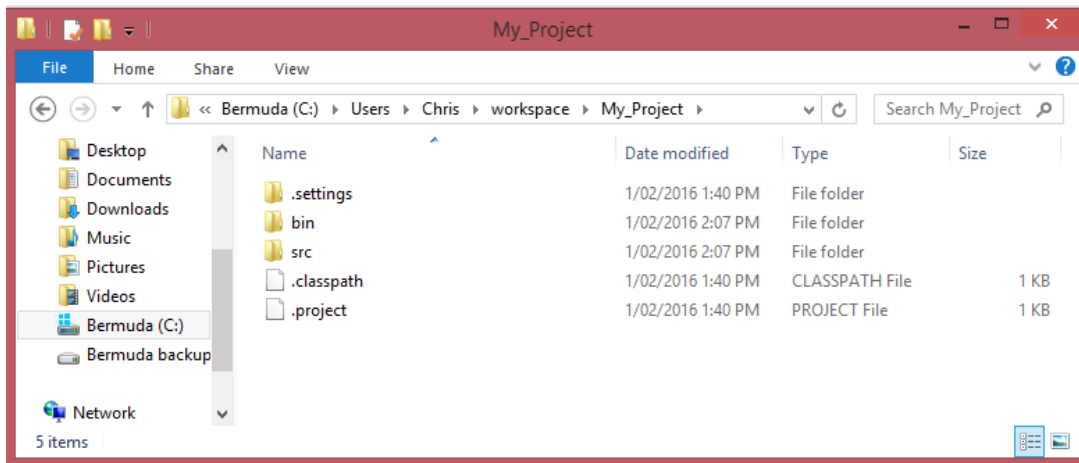
Figure 15: Project Output

Figure 15 shows several other folders. The .settings folder is used to store all of the configuration settings for the project in a preferences file that defaults to *org.eclipse.jdt.core.prefs*. The bin folder is used to store the binary application output, which is the executable machine code and other required resource files that the Java compiler produces from our source code.

Debugging and running applications

Let's test our application. To debug an application is to run it in such a way that we are offered extra information that will help us track down and eliminate bugs. Typically, when we develop a Java application, we test the program at intervals in order to make sure everything is running smoothly. As our projects become larger, we must examine all of the interacting components in great detail in order to be certain that we have not done anything wrong.

Eclipse allows us to debug an application, but it also allows us to run the application. Running the application is like debugging, except that we get less information as to what's happening within the application. When we use Run, the compiler (javac.exe) might optimize our code and remove all debugging information. It is recommended that while an application is being developed, we use Debug. When you are satisfied that your application is complete and working properly, you can use Run to build your application's first release.

In order to debug your code in Eclipse, click **Debug** (it has a beetle icon, as in Figure 16). The Play button (to the right of the debug), is for running your application without debugging.
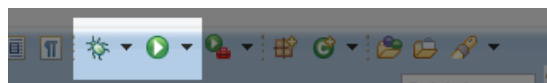
Figure 16: Debug and Run Buttons

💡 *Tip: There are many configurable shortcut keys available in Eclipse. The shortcut key for Debug is F11. The shortcut key for Run is Ctrl+F11.*
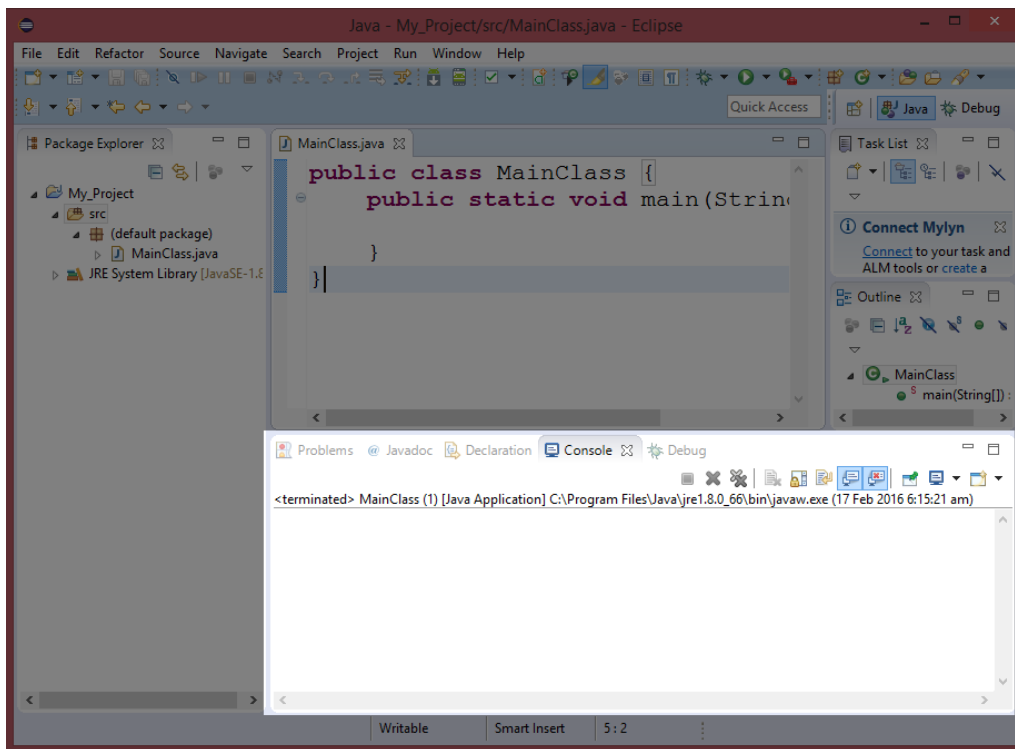
Figure 17: Output Console

When we run or debug our application, the output of the application will appear in a window at the lower portion of the Eclipse screen. Notice that there is a collection of tabbed pages in Figure 17, including a tab named Console. Our current program does nothing, so this Console tab is blank after we debug or run the application from Code Listing 2.0, but in future chapters we will print text to this console and read data from the keyboard.

Now that we have examined how to start a new application, how to add a class, and how to add the main method as an entry point to our program, we can start programming Java to perform tasks for us.

Chapter 3 Writing Output

Code Listing 3.0 shows the code to a basic "Hello, World!" application. The program prints some text to the console, then it exits. The numbers on the left side are included for reference and should not be typed into your code. Type or paste this code over the code in the MainClass.java file we created earlier.

Code Listing 3.0: Printing Output

```
1: public class MainClass { // Definition of MainClass
2: public static void main(String[] args) { // main method entry point
3:     System.out.println("This is a test application!");
4:     } // Close brace for main method.
5: } // Close brace for MainClass.
```

When we execute this code with either Debugging or Running (use keys F11 or Ctrl+F11, or click the debug or run buttons), the program will print "This is a test application!" to the system output.

Line 1: public class MainClass. This line declares a class called MainClass. The class is marked with the public modifier, which means it is visible to all other entities and packages in the application. After the public modifier, we use the keyword class, then we supply a unique name for our new class—MainClass, in this instance. This is followed by a curly brace, {, that indicates the beginning of the body of the class. Line 5 contains the matching closing curly brace, }, that marks the end of the body of this class.

Line 2: On line 2 we declare a static member method called main. The method is marked public, so that classes outside of this one are able to call it. The method is marked static, which means it belongs to the class rather than to any particular instance of the class. The method does not return any data to the caller, so the return type is void. In the brackets, the (String[] args) is the parameter list. This means an array of strings will be passed to this method when it is invoked. The array of strings is the command-line argument—when we run a program from the command line, we can optionally supply extra arguments. We will not be using the command line, and this array will always be empty in the example programs. After the parameter list, we see an open curly brace, {, that marks the beginning of the body for this method. The matching close brace for the main method is on line 4.

Line 3: This line invokes the println method, which belongs to the System.out class. System.out is a class, and println is a public static method belonging to the class. The method takes a String parameter and prints the string to the console output. The string is enclosed in brackets that indicate that it is a parameter to the method call. The parameter is a String because it is enclosed in double quotes. The line ends with a semicolon ";" that indicates that the statement is finished. All statements in Java end with a semicolon.

Line 4: This is the matching closing brace for the main method. The opening brace appeared on line 2. This brace indicates that we have finished describing the method.

Line 5: This is the matching closing brace for the `MainClass` class. The opening brace appeared on line 1. This brace indicates that the description of the `MainClass` is finished.

Don't worry if some of these previous descriptions are too brief. We will look further into each of the keywords and other mechanisms as we progress through this book. Initially, Java has a fairly steep learning curve. In order to begin programming Java, even the most basic program contains a lot of syntax (syntax is the grammar of a programming language—the rules, spelling, and punctuation).

The name of the class can be any valid identifier. Valid identifier names contain letters and digits, but they cannot begin with a digit. Identifiers must be a single word (no spaces, tabs, or punctuation of any kind is allowed), but you are able to use the underscore character, e.g., `Main_Class` is a legal identifier, but "`Main Class`" is not because it contains a space.

When you run or debug the application from Code Listing 3.0, you will notice that message is printed to the Console panel at the lower portion of Eclipse, as per Figure 18.
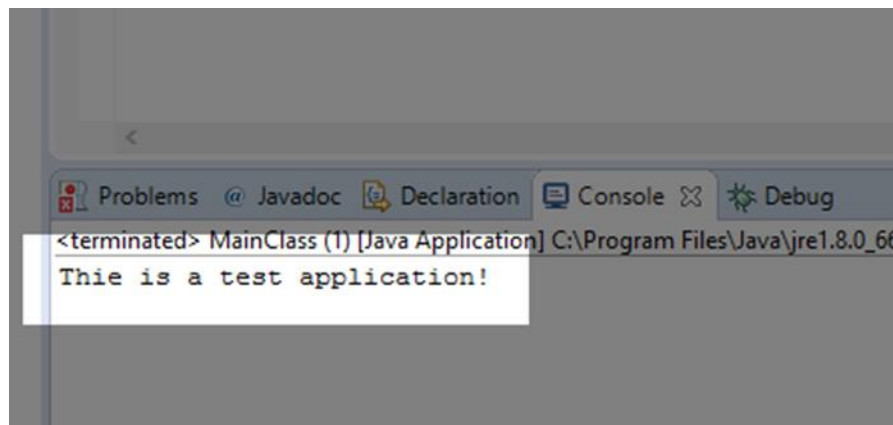
Figure 18: Code Listing 3.0 Output

Foundations of Java

In this section, we will briefly look at several aspects of programming Java that will become second nature as you become more accustomed to the language.

White space

White space is the use of space within code. This includes new lines, spaces, and tabs. Generally speaking, the Java compiler ignores white space, and programmers can lay out code however they wish. There are exceptions to this rule—for instance, in Strings, the compiler preserves the spaces because there is a big difference between the two strings: `This is a variable` and `Thisisavariable`.

```
c.setRadius(100);

c .         setRadius

                (       100

    )

                        ;
```

Code Listing 3.1 shows two calls to the same method: `c.setRadius(100);` (this is a code snippet and has no `main` method—this code will not execute as an application). The compiler will read these two calls as being identical even though the second has a lot of white space and is very difficult to read.

Although white space is generally ignored by the compiler, we cannot break up tokens. The symbols used to express elements of the language are called tokens, and if we place white space between the characters of tokens, the compiler will not understand what we mean. For example, the term `setRadius` in Code Listing 3.1 is a token. If we place a space between the two words to create `set Radius`, it no longer forms a single token, but rather two. Likewise, we cannot place white space in the middle of numerical literals. For example, `150` is a single token meaning *one hundred and fifty*, but `1 50` is two tokens meaning *one* and *fifty*.

The use of white space is closely tied to programming style. There is no correct or best way to set out code, and no best way to use white space. For the code in this e-book, I have adopted common conventions used by many Java programmers that create code that is easy to read and debug. If you are new to programming, try to match the bracing and tabbing I have used here. This style of bracing and tabbing is very common, and if you are familiar with coding and reading it, you will have little trouble understanding other programmer's code.

Tip: Learn to read and program in any style. How best to use white space is a hotly debated topic in many programming communities. But, as programmers, we need to read and understand other people's code, and we often need to program in teams in which the use of white space must be the same for all programmers. A programmer who can read anybody's code and can flip to any style without hesitation is far more valuable than one who argues about tabs versus spaces.

Identifiers

An identifier is a name a programmer selects to represent some entity or mechanism in the code. We select identifier names for our variables, methods, classes, and objects. In Code Listing 3.1, the terms `MainClass`, `main`, and `args` are all identifiers. `System`, `out`, and `println` are also identifiers—they were named by the folks who programmed the Java libraries.

When we select an identifier, we must ensure that it is unique to the current scope (or unique to the current code block—we will look later at scope and code blocks in detail). We cannot name two variables `myVariable` in a single code block because the compiler has no way of knowing which variable we are trying to reference (however, we will see that there is a scope operator that enables us to bend this rule slightly!).

We are not allowed to name our identifiers after keywords (see the next section for a description of keywords). For instance, we cannot name our methods or variables `if` or `while`. Again, this comes down to the fact that the compiler needs to differentiate keywords from identifiers, and it cannot do this if the identifiers have the same names as the keywords.

Identifier names cannot contain any white space (this includes spaces, tabs, and new lines). Identifier names must be a single, unbroken word. Identifier names can contain letters, digits, the dollar sign, $, and underscores, "_", but they cannot begin with a digit. The compiler sees any token beginning with a digit as a numerical literal, and it will not allow us to name our identifiers things such as `67Hello` because this would mean the number 67 with the meaningless suffix Hello.

Identifier names are case sensitive, which means the identifiers `patientData`, `PATIENTdata`, and `patientdata` are all seen as distinct by the compiler. Naming multiple identifiers that differ only in case is not recommended because this can lead to confusing and difficult-to-read code, although the compiler will have no trouble differentiating the identifiers.

Example identifiers are: `someVariable, i, the_timer_for_rendering_graphics, iterator15, MY_CONSTANT, $euclideanDistance`.

In Java, developers often use a technique called Camel Case for naming our identifiers. In Camel Case, the first letter of each of word is uppercase—except for the very first letter. For instance, a function for computing a score might be called `computeScore`, and a variable for summing a total of ages might be called `sumTotalAges`.

An exception to this rule of thumb is naming classes. When naming classes, Java programmers often use uppercase for the first letter of all words, including the first. A class for holding an automaton's generation grid might be called `AutomatonGenerationGrid`. This makes variables and classes easy to distinguish in the code. In Code Listing 3.1, the identifiers `MainClass`, `System`, and `String` are all classes, which means they begin with an uppercase letter. Whereas the identifiers `args`, `out`, and `println` are variables and method names, so they begin with a lowercase letter.

In addition to Camel Case, it is common to name final variables (similar to constants in other languages) with all capital letters. A final variable is simply a named value that cannot be changed, such as the mathematical variables `PI` or `E`. Because programmers use uppercase for all letters, when naming constants, it is common to split the words by using the underscore for spaces. A final variable for seconds per hour might be called `SECONDS_PER_HOUR`, or a final variable for the number of days in June might be called `NUMBER_OF_DAYS_IN_JUNE` or simply `DAYS_IN_JUNE`.

One final rule of thumb when naming identifiers—identifiers should be named logical, descriptive names in order to ensure that the code remains as readable as possible. For example, it is generally a better idea to name a variable `marginOfError` rather than `mOE` or `m;`,

and it is better to name a function `computeMedianOfHouses` rather than `doThing` or `cabbagesWithWhippedCream` (assuming the function computes the median of houses). If we do not name our identifiers with logical, easy-to-understand names, other programmers might have difficulty comprehending and maintaining our code. And worse—we will not understand our own code, either. It is surprising how little time it takes before our own code becomes gibberish unless we are careful and name our identifiers intelligently.

Note: There are a few exceptions to naming identifiers with logical and descriptive names. For instance, it is common to call iterators (counter variables) in for loops single letter names such as `i`, `j`, `x`, `y`, or `z`. This provides no indication as to what the variable means, but their purposes will still be understood by other programmers. Also, some algorithms have specific variable names in their descriptions, and the common names for the variables are often used when programming an implementation of the algorithm, such as the A* path-finding algorithm, which uses the names `t` and `h`.

Keywords

Computer programming languages are very simple compared to human languages such as English. Java has a small collection of grammatical symbols (such as { and } braces) and a small collection of nonlexical terms called keywords. And Java has a series of rules for how to use these elements (i.e. it has a grammar). In Eclipse, we can identify keywords in our code because they are highlighted in dark purple, and they are bold, such as **class** and **public** in Code Listing 3.1.

Keywords form the main backbone of the way Java works. Keywords are reserved for specific uses in Java, and they cannot be used as identifier names. See the Appendix for a reference on the available keywords. The following is a list of the keywords we have encountered so far:

public: The public keyword is an access modifier. It means that external objects (objects that are not part of the class being defined) can access the methods or member variables.

class: The class keyword is used to indicate that the code block that follows is a description of a class.

static: The static keyword marks member methods and variables as belonging to the entire class rather than specific objects or instances of the class. This means we can access static members without creating objects from the class.

void: The void keyword is used as a placeholder for a return type when a method does not return any data. All methods must have a return type or must have the return type of void. The only exception to this rule is constructors that have no return type and have the same name as the class.

Dot operator

In Code Listing 3.1, line 3 reads as follows: `System.out.println("This is a test application!");`. This line contains several periods (or full stops). This is the dot operator (sometimes called the scope operator). It is the operator that connects classes to methods and

member variables. `System` is a class, and `out` is an entity belonging to the `System` class. When we supply the terms `System.out` connected by the dot operator, we are saying, "The out entity that belongs to the `System` class" or, in short, "`System's out.`"

The method `println` belongs to the `out` entity. When we invoke the method by specifying its name, `System.`*`out`*`.println`, followed by a parameter list, (`"This is a test application!"`), we are really saying: Execute the code specified in the method called `println`, which belongs to the entity `out`, which belongs to the class `System`. Notice how the dots indicate ownership from left to right—`System` owns out, and out owns `println`.

We will later look in more detail at object-oriented programming, but the main objective of object-oriented programming is to define hierarchies of objects, each owned by and potentially owning more objects, and each capable of performing methods with a collection of member variables. Different classes can contain variables and methods with the same names. For example, we might have multiple variables called `cost`, one belonging to a class called `House` and another belonging to a class `Vehicle`. We can distinguish these variables by using the dot `House.cost` vs. `Vehicle.cost`.

Comments

Code Listing 3.2 shows several comments, with the Eclipse highlights comments in green. A comment is a note to the programmer. The computer will ignore all comments when the program executes. There are many applications for comments, and every programmer has a unique style of commenting on his or her code. Although comments do not change the way a program runs, developing good commenting skills is extremely important. Comments allow a programmer to describe exactly what the code should be doing. This means that any other programmer should be able to read the comments and understand the implications of complex code at a glance.

In Java, there are two types of comments—single-line comments and multiline (or block) comments (see Code Listing 3.2).

Code Listing 3.2: Comments

```java
// This is a single-line comment!

System.out.println("Press a key to continue..."); // Single-line comment!

/*

This is a multiline comment. It extends from the opening symbol of a slash and

star to the closing symbol, a star followed by a slash!

*/
```

Code Listing 3.2 shows the two different types of comments. The first is the single-line comment. These begin with two slashes, //. All text to the right of the // is ignored by the computer. If a line of code begins with the //, the entire line is ignored, but you can also place the // after some statements, such as a `println`.

The multiline comment begins with /* and extends until the */. The programmer can place any text inside the /* and */, including new lines. This is sometimes called a block comment because it enables a programmer to write blocks of text or to comment out blocks of code.

In the upcoming code examples, we will see many comments included to help the reader identify what the different lines of code are supposed to do.

Challenges

Challenge 3.0: Change the identifier `args` so that it is called `arguments.` Does the program compile and run?

Challenge 3.1: Change the identifier `main` to `mainMethod.` Does the program compile and run?

Challenge 3.2: Change the message that is printed to the screen to the string "`Hello, World!`".

Challenge 3.3: Change the name of the class `MainClass` to `MyClass.` Does the program compile and run?

Challenge solutions

Challenge 3.0

The program compiles and runs when the `args` identifier is changed to `arguments` because this identifier is simply a parameter to the `main` method. Its name is up to the programmer.

Code Listing 3.3: Challenge 3.0

```java
public class MainClass {

public static void main(String[] arguments) {

    System.out.println("This is a test application!");

    }

}
```

Challenge 3.1

The program does not compile and run if we change the `main` method's name to `mainMethod`, because the JVM needs an entry point to the program and it searches for a method called `main`. When it cannot find such a method, it will not run the application.

Code Listing 3.4: Challenge 3.1

```java
public class MainClass {

    public static void mainMethod(String[] args) {

            System.out.println("This is a test application!");

        }

}
```

Challenge 3.2

Code Listing 3.5: Challenge 3.2

```java
public class MainClass {

    public static void main(String[] args) {
```

```
            System.out.println("Hello, World!");

    }

}
```

Challenge 3.3

The program will not compile and run if the name of `MainClass` is changed to `MyClass` without the name of the file (in which the code is written) also changing. In Java, classes should be placed into source code files with the same names.

Code Listing 3.6: Challenge 3.3

```java
public class MyClass {

    public static void main(String[] args) {

            System.out.println("This is a test application!");

    }

}
```

Chapter 4 Reading Input

Reading System.in

The program in Code Listing 4.0 illustrates how to read input from the user. Specifically, we ask users for their names and to read some text that they type at the keyboard.

Code Listing 4.0: Reading Input

```
1: import java.util.Scanner;
2: public class MainClass {
3:    public static void main(String[] args) {
4:         Scanner scanner = new Scanner(System.in);
5:         System.out.println("What is your name?");
6:         String str = scanner.nextLine();
7:         System.out.println("Nice to meet you, " + str + "!");
8:    }
9: }
```

Line 1 imports the `Scanner` class from the `java.util` package. A `Scanner` scans a stream, waiting for and reading data. There are literally thousands of prewritten tools included with Java that we can `import` in a similar way if we wish to use them. Code Listing 4.0 uses a `scanner` to scan `System.in`, which is a stream of bytes representing the data the user types to the console.

Line 2 is the opening declaration for the `MainClass`.

Line 3 is the opening declaration for the `main` method.

Line 4 is where we declare a `java.util.Scanner` object. We specify that our `scanner` is to scan the `System.in` stream. The `Scanner` class has a constructor that takes a stream as a parameter (a constructor is simply a special method used to create an instance of a class). A `scanner` object (called `scanner` with a lowercase s because it is an object, not a class) is created when we use the `new` operator to call the constructor. We will look at exactly what this line does in a lot more detail when we cover the `new` operator in the chapter on object-oriented programming. For the moment, this line means create a `Scanner` object called `scanner` and have it scan the input stream `System.in`.

Line 5 prints a prompt to the console asking for the user's name.

Line 6 is where we read text from the keyboard. We do this by calling the `scanner` object's `nextLine` method. The method takes no parameters, so we supply empty brackets (and). This line will read text until the newline symbol is read (i.e. until the user hits enter or return on the keyboard). `Scanner.nextLine` will return the text the user typed as a `String` that we assign to the `String` variable `str` with the assignment operator `String str =`.

Line 7 prints out the string "`Nice to meet you, Chris!`" in which Chris is replaced with whatever name the user typed at the prompt. Notice how the string "`Nice to meet you,`" has the variable `str` added to the middle by closing the double quotes and employing the string concatenation operator +. This is an example of how we join or concatenate two or more strings together.

Line 8 is the matching closing brace for the `main` method's code block. At this point, the object `scanner` and the `String` variable `str` fall out of scope (this means they no longer exist). Programs begin at the start of the `main` method, and they end at the closing brace of the `main` method. So this line is really the close of our application. We can also use the `return` keyword in the body of the `main` method in order to return from the method, although note that this would also terminate the application. We will look more at the `return` keyword when we look at defining methods.

Line 9 is the matching closing brace for the `MainClass` class.

In order to run the program, we can use the Debug or Run buttons. We will see that the program halts after it prints the message "`What is your name?`". In order to input a string, click in the Console tab at the prompt. You should see a blinking cursor. Type your name (or any other string) and hit enter. After you hit enter, the `scanner` object has a line of text to read, and, where we set the value to the `str` variable, it will return this to the caller. (Figure 19 shows an example interaction with the program in Code Listing 4.0, in which the green text is what I typed at the prompt).
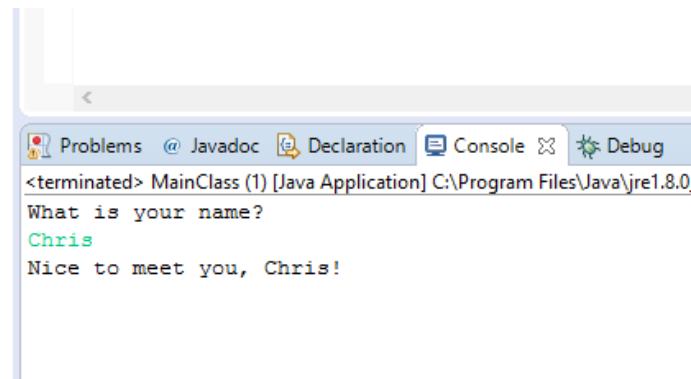

Figure 19: Code Listing 4.0 Output

In this e-book, *Java Succinctly Part 1,* we will concentrate on the core mechanisms of the Java language, and we will be restricted to this Console for input and output. We can create fully functional graphical user interfaces with Java, and in *Java Succinctly Part 2* we will look at how to do this.

Warnings and errors

When we debug or run the program in Code Listing 4.0, we will see that Eclipse gives us a warning. A warning is a potential problem with our code, but one that does not prevent the code from executing. Eclipse can detect two types of problems in our code: warnings (which do not

prevent the code from running, but which should probably be fixed) and errors (which are problems that prevent the code from running).

Generally, if we have any warnings in our programs, we should fix them because they can lead to unwanted behavior later if they go unchecked. In the next program, we will see two extra lines of code that are designed to close the scanner and fix the warning that Eclipse is highlighting.
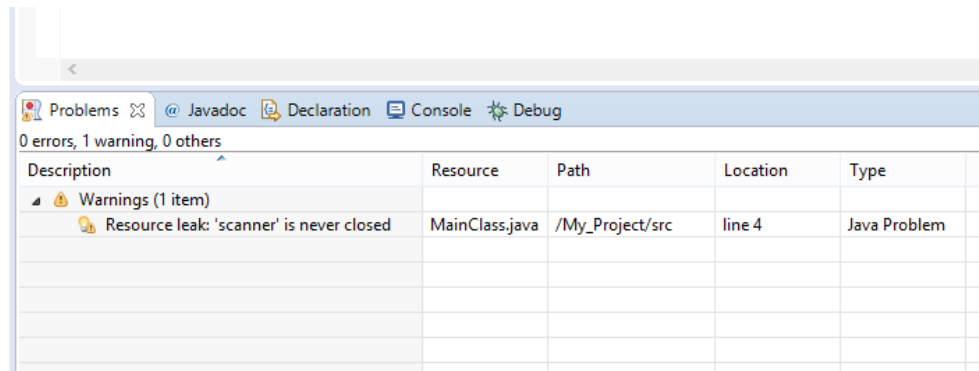

Figure 20: Warning

Figure 20 shows the warning that our code has generated. This warning will only be visible when we attempt to debug or run our program. The warning tells us that the problem is `'scanner' is never closed`. It tells us that the problem is in the file `MainClass.java`, and it occurs on line 4. If we double-click this warning in the Problems panel, Eclipse will take us to line 4 of the `MainClass.java` file so that we can fix the problem or gain some understanding as to what is causing it. Figure 21 shows an example of an error.
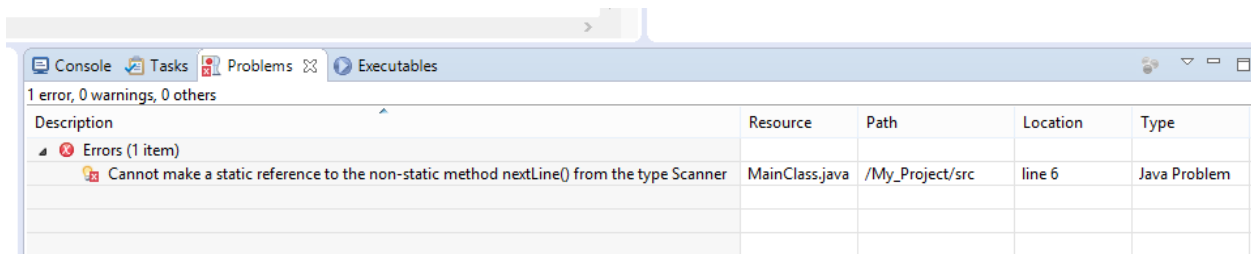

Figure 21: Error

In order to cause this error (Figure 21), I purposely changed line 6 of Code Listing 4.0 to read `String str = Scanner.nextLine();`. Note the capital `S` in `Scanner`. This small change is a mistake, and it means that the program will not compile and run. Looking at the Problems panel, we get a cryptic error message that explains that `nextLine` is not `static`, and we are attempting to use it as if it was. Error messages are often very difficult to understand, but a solid foundation in the basics of Java will help you make sense of them. Essentially, this error is saying that we need to use the object `scanner` rather than the class `Scanner` if we wish to call the method `nextLine()`. Otherwise, the program will not compile and run.

Eclipse perspectives

A perspective is a layout of windows and panels in Eclipse. By default, Eclipse is set to the Java perspective (Figure 22). But when we debug our code, or when our program fails to compile due to errors, Eclipse will switch to the Debug perspective (Figure 23). The Debug perspective offers extra panels of information that might help us track down and fix bugs (later we will look at watching variables and stepping through code in the Debug perspective).
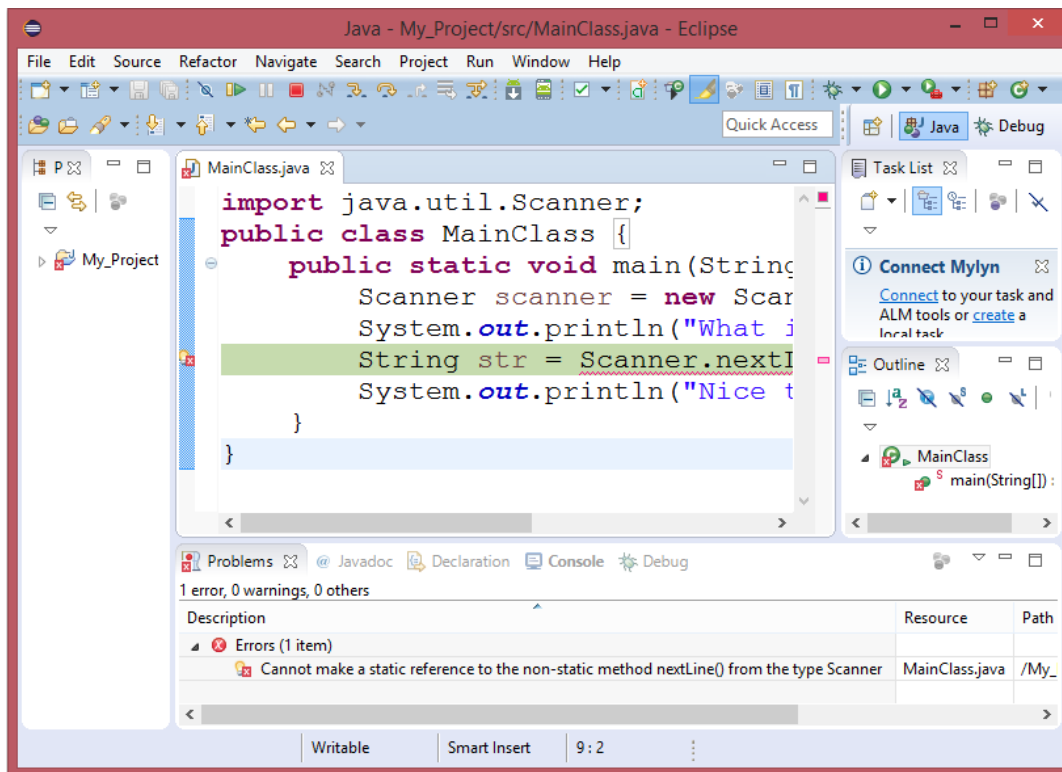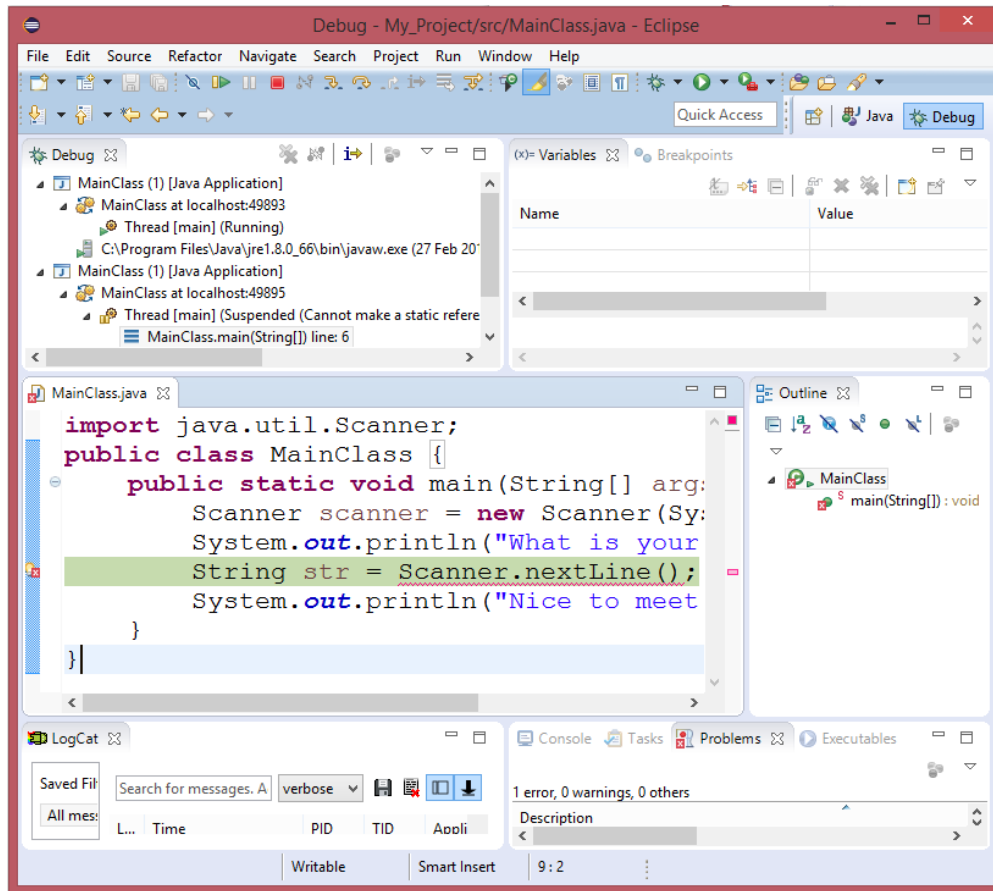

Figure 22: Java Perspective

Figure 23: Debug Perspective

In order to switch between the two perspectives, we use the two perspective buttons in the upper-right corner of Eclipse (marked Java and Debug, as in Figure 24).
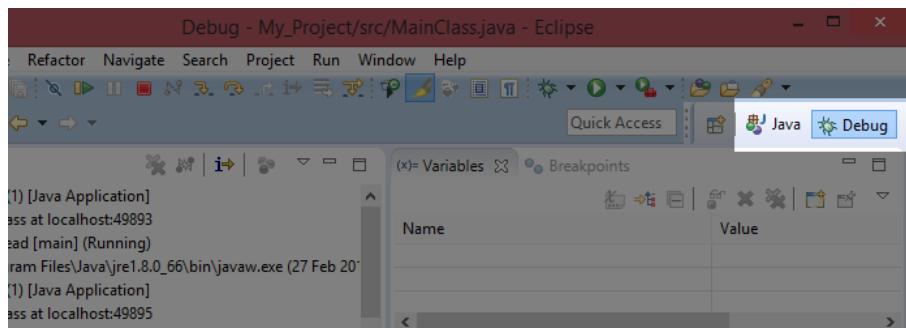
Figure 24: Java and Debug Perspective Buttons

When we debug or run our program and Eclipse pauses it, it will take us to the Debug perspective, but our program will not shut down. In order to stop the program (shut it down completely, so that it is no longer running), we need to press the Stop button, which is in the main control panel of both the Java and Debug perspectives (see Figure 25). If we do not stop our applications, they will continue to take up resources in the system.
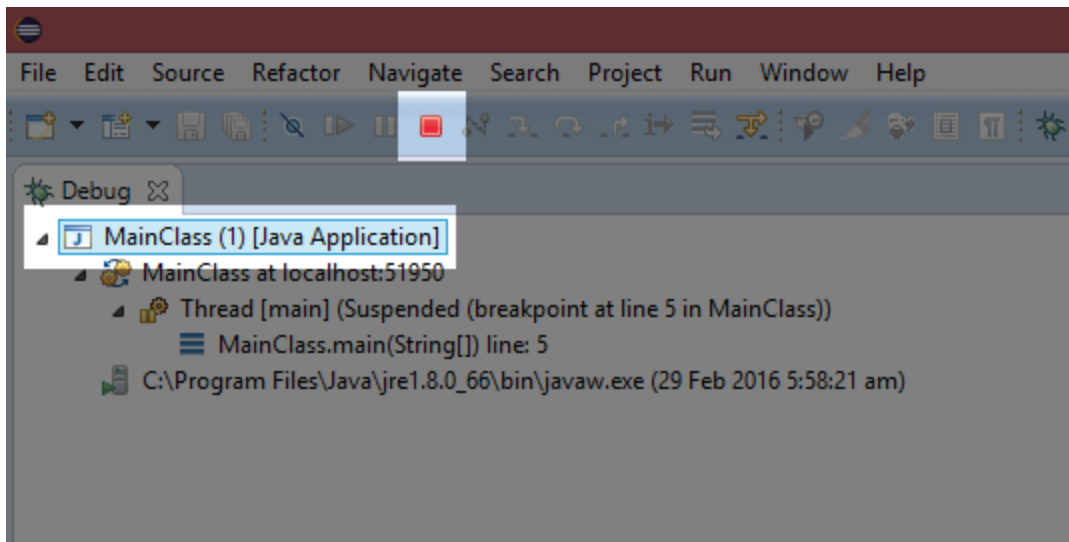
Figure 25: Stopping a Running Application

Figure 25 shows the Stop button highlighted when Eclipse has paused our application. When you are in the Debug perspective, the Debug panel on the upper-left side of the screen shows all the running instances of our application (in Figure 25, there is only one instance). If you have multiple instances running, you can close each individually by selecting it in the panel, and clicking the Stop button.

Reading multiple values

Our next example code illustrates how Java code is usually executed in order, from top to bottom, one line after the other. Program 3 asks the user for both a name and a password. The program will not proceed past the first occurrence of `scanner.nextLine()` until the user has input a line of text at the console.

Code Listing 4.1: Reading Multiple Strings

```java
import java.util.Scanner;
public class MainClass {
     public static void main(String[] args) {
          // Declare all variables.
          String userName;
          String userPassword;
          Scanner scanner = new Scanner(System.in);

          // Request the user's name.
          System.out.print("What is your name? ");
          userName = scanner.nextLine();

          // Request the user's password.
          System.out.print("Ok, and what is your password? ");
```

```java
            userPassword = scanner.nextLine();

            // Print out the info we read in a message.
            System.out.println("Whoa! That was too easy! You're crazy, " +
                        userName + ", you just told a complete stranger " +
                        "that your password is " + userPassword + "!");

            // Close the scanner to avoid memory leaks.
            if(scanner != null)
                scanner.close();
        }
}
```

Code Listing 4.1 shows some common conventions. First, I have declared all the variables at the beginning of the method (we will look at variable declarations in the next chapter). And, at the end of the method, I have called a method called "close" to ensure my `scanner` object does not cause any memory leaks (these two lines of code fix the warning we were getting in the first program of this chapter). I have also used an `if` statement (we will look at method calls and `if` statements shortly). The program illustrates that we can potentially read as many strings from a scanner as we like and print out prompts between each.

I have used `System.out.print` in place of `println`. The `print` method prints the strings in the same way as `println`, except that it does not place a new line at the end of the string. Figure 26 shows an example interaction with the program from Code Listing 4.1. Notice how the user input (the green text) is on the same line as the prompt. Had we used `println`, the green text would be on its own line.
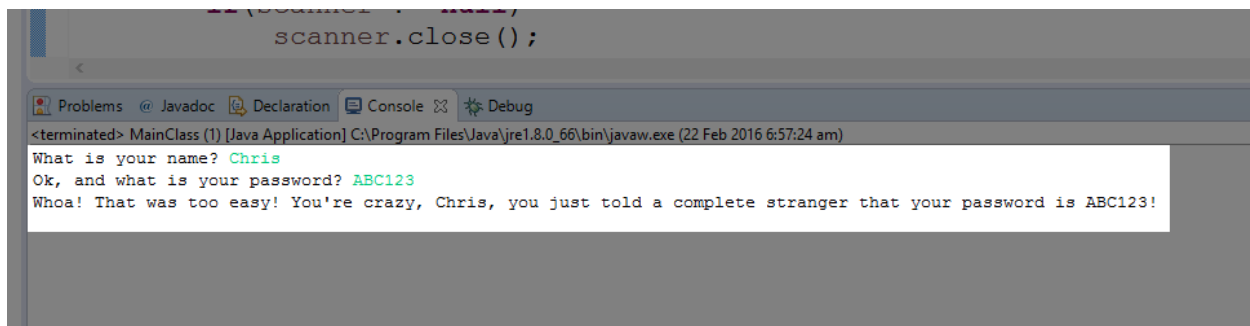

Figure 26: Example Output from Code Listing 4.1 Code blocks and curly braces

The following program is intended as a study in code blocks. Java is one of many languages that inherits from a very old language called C. The languages are collectively called curly brace languages because they use the curly braces { and } to designate blocks or areas of code (these languages are also sometimes also called C-like languages). Each opening brace, {, has a matching closing brace, }, and each pair contains one or more lines of code between, which is called a code block. You can open and close a code block for no reason at all almost anywhere in Java source code, but code blocks are usually a mechanism for joining together several lines of code into a unit for some specific purpose. The specific purposes are generally to define the bodies of loops, methods, and classes.

Code Listing 4.2 shows a program with the code blocks highlighted. This program is much more complicated than the programs we have seen so far. The details of how this program works will become apparent by the time we look at methods in a future chapter. For now, the program is meant to illustrate matching { and } braces and nested code blocks.

Code Listing 4.2: Code Blocks

```java
public class MainClass {

    // Method to return lowest prime factor of the integer x.
    private static int GetLowestFactor(int x) {
        int sqrt = (int)Math.sqrt(x);
        if(x % 2 == 0) {
            return 2;
        }
        for(int q = 3; q <= sqrt; q+=2) {
            if(x % q == 0) {
                return q;
            }
        }
        return x;
    }

    // Determines if x is prime by seeing if it is equal to
    // its own lowest factor.
    private static Boolean  IsPrime(int x) {
        return GetLowestFactor(x) == x;
    }

    // This program lists the prime numbers in the range 1 to 1000.
    public static void main(String[] args) {
        // Declare local variables.
        int i = 2, count = 0, countPerLine = 10;
        // Show a description of the program.
        System.out.println("Prime numbers from 1 to 1000 are: ");
        // Loop that finds the primes.
        while(i <= 1000) {
            // If i is prime...
            if(IsPrime(i)) {
                // Add it to the list...
                System.out.print(i + ", ");
                // Increment the count.
                count++;
                // If there's countPerLine items on this line,
                // print a new line.
                if(count % countPerLine == 0) {
                    System.out.println();
                }
            }
        }
```

```
                    // Increment i to the next number.
                    i++;
            }
        }
}
```

If you copy and paste the code from Code Listing 4.2 into Eclipse, you should get a list of the prime numbers between 1 and 1000. In Code Listing 4.2, each pair of matching braces has been highlighted in a different color. Notice that code blocks are nested inside other code blocks. That is—smaller code blocks begin and end within the braces of outer code blocks. The yellow braces surround the body of the `MainClass`, and the `MainClass` contains the code blocks for several methods, each of which contains its own code blocks.

Notice also that code blocks can be created for many different mechanisms—classes, loops, If statements, and methods all have associated code blocks depicted in Code Listing 4.2.

We will now take a brief detour into a completely different but very useful aspect of programming—debugging code.

Detour: debugging code

The ability to debug and maintain code is one of the most important attributes a programmer can have. With that in mind, when you first test your program in Eclipse, select the Debug icon, instead of the Run icon.

Printing out values

We often need to know the value of variables while our program runs. The most basic form of discerning the value of a variable is to output this value to the screen. We can use a `System.out.println` to do this, such as `System.out.println("The value of i is " + i);`. This is a very primitive method for debugging, but in many circumstances it is all we will need. Generally, it is better to place a breakpoint and a watch on the variable, as described in the following section.

Setting breakpoints

A breakpoint is a position in the code at which we want the program to pause and give us control so that we can examine exactly what is happening in the code. When a program pauses at a breakpoint, we can inspect the values of variables and step through the code—one line at a time. To set a breakpoint, right-click in the margin of the code view in Eclipse. This will bring up a context menu with the option Toggle Breakpoint, as per Figure 27.
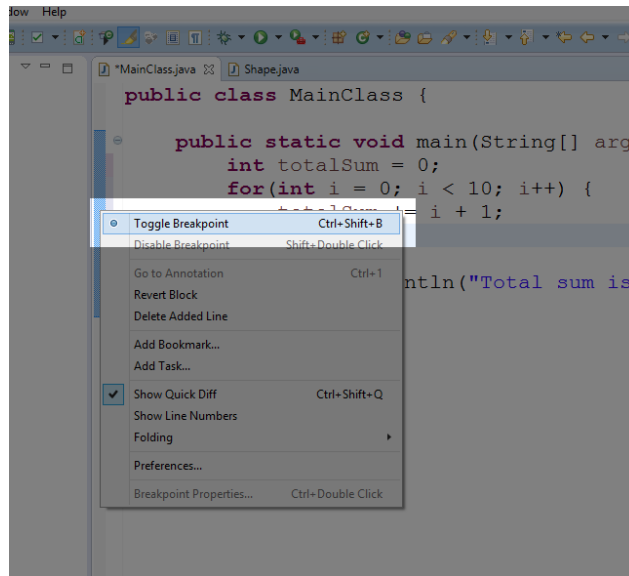
Figure 27: Toggle Breakpoint

When a breakpoint is active (i.e. if the program will pause when it reaches the breakpoint), you will see a small breakpoint symbol in the margin of the code window, as illustrated in Figure 28.
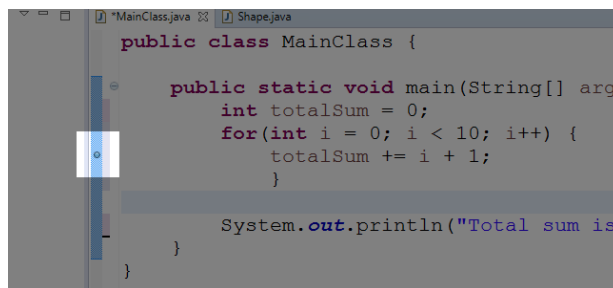

Figure 28: Active Breakpoint

When the breakpoint indicator is blue, as in Figure 28, it means the breakpoint is active. You can disable a breakpoint by right-clicking the breakpoint symbol and selecting Disable Breakpoint from the context menu. When a breakpoint is disabled, the circle is white and the program will not pause when it reaches the breakpoint.

Debug perspective in Eclipse

When the program reaches an active breakpoint, it pauses and gives us control. The first time you break in Eclipse, you might get a message box, as shown in Figure 29. Eclipse is designed to offer several extra tools specific to debugging, and it offers to change the view to the Debug perspective. Click the box marked "Remember my decision" and select "Yes" to enter the Debug perspective.
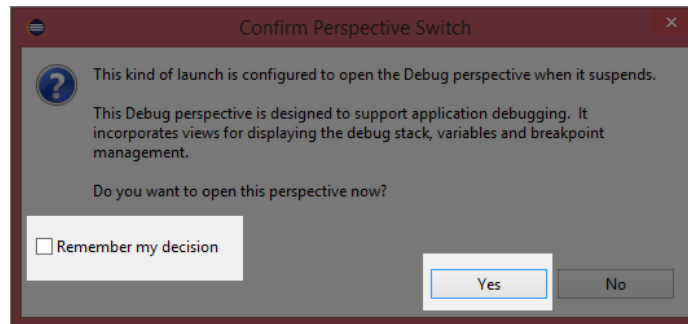
Figure 29: Confirm Perspective Switch

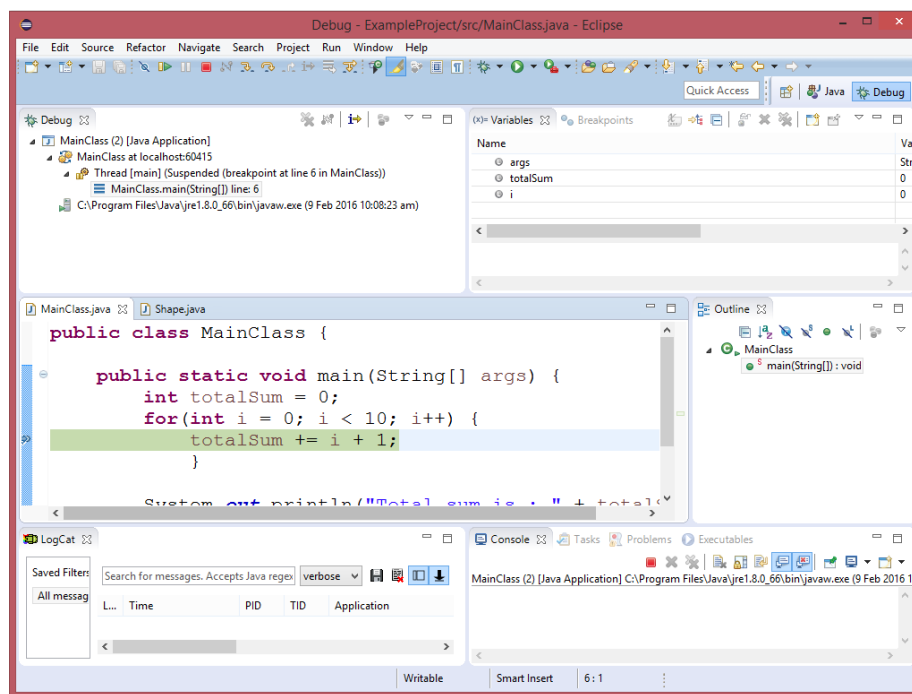The Debug perspective is illustrated in Figure 30.

Figure 30: Debug Perspective

Notice the line highlighted in green in Figure 30. This is the line that the program is ready to execute. At the top of the Eclipse screen, you will see a green Play button, a Pause button, and a Stop button. In order to stop the program from executing, click Stop. In order to resume normal execution (i.e. to begin executing again until the program finds the next breakpoint), click Play. And, if your program is executing already and has not hit a breakpoint, you can pause it without adding a breakpoint by using the Pause button (see Figure 31). Depending on the circumstance, some or all of these buttons may be greyed out (for instance, if the program is already paused at a breakpoint, the Pause button is not available).
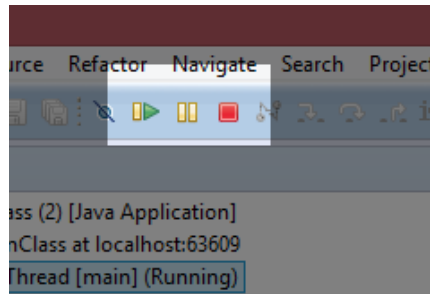
Figure 31: Play, Pause, and Stop Buttons

💡 ***Tip: You can skip all breakpoints by clicking the button to the left of the Play button in Figure 29. This is handy when you have set a lot of breakpoints in order to examine the exact flow of the program, but you want to test the overall functionality without permanently disabling the breakpoints.***

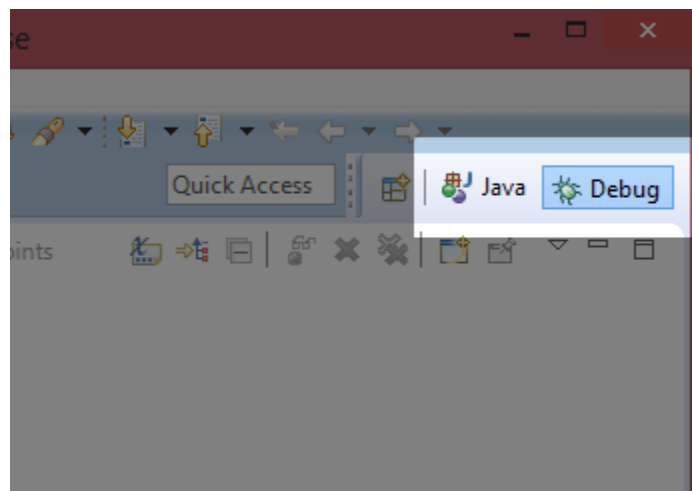You can switch between the Java and Debugging perspective by clicking the buttons in the top-right corner of Eclipse, as illustrated in Figure 32.

Figure 32: Changing the Perspective of Eclipse

Stepping Over and Into code

Stepping through code can be done in two ways:

- Step Over
- Step Into

When the program is paused, we can look very closely at exactly what is happening to all the variables while it executes one line at a time. We do this by employing the Step Over and Step Into buttons (see Figure 33). You will find these buttons at the top of Eclipse when the program is paused.
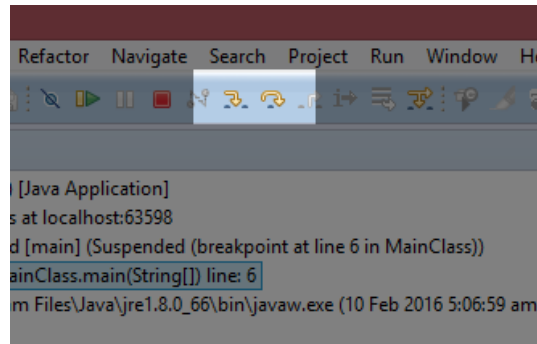
Figure 33: Step Over and Step Into Buttons

Step Over means the program will execute the current line of code, then pause again. It will not jump into method calls. Instead, it will execute the method as a single step. Step Into means the program will execute the next line, but if the line is a method call, it will jump to the body of the method and pause. The shortcut key for Stepping Into is F5, and the shortcut for Stepping Over is F6.

Watching variables

As we Step Over and Step Into our code, we can view exactly which values our variables contain by looking in the Variables tab, as in Figure 34.
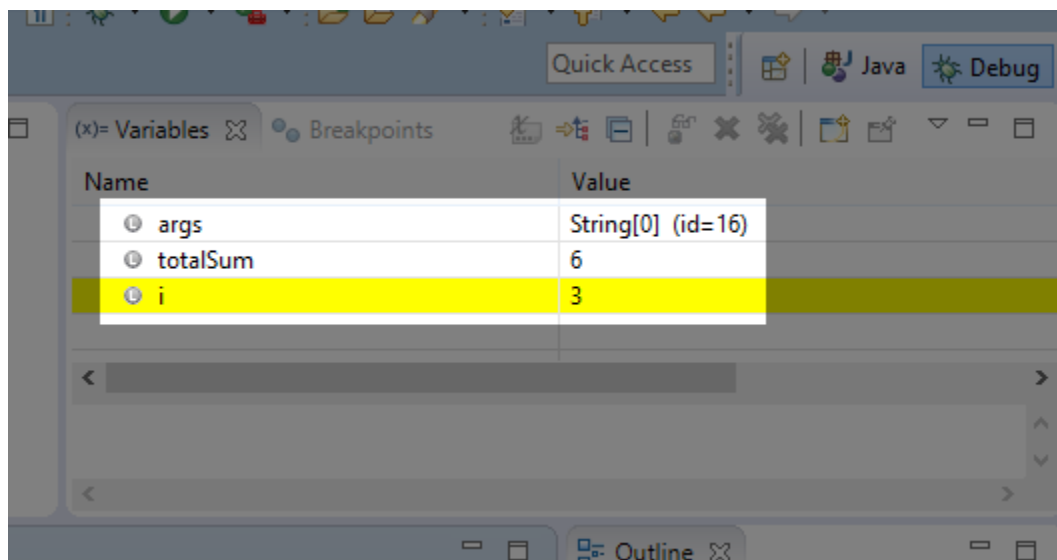
Figure 34: Variables Tab

When the application is paused, the Variables tab shows all the variables that are presently in scope. The variables highlighted in yellow have changed during the last Step Over or Step Into.

This has been a very brief tour of some of Eclipse's debugging tools. They can greatly help us track down problems in our code, but they can also be used to teach us how Java works. Later, when we look at control structures, I strongly recommend that you set breakpoints in the

programs and step through the code, watching the way the program branches and executes loops. There is no better indication as to how Java works than looking at it one line at a time.

Challenges

Challenge 4.0: Change the program in Code Listing 4.0 to ask for the user's age instead of name.

Challenge 4.1: Change the program in Code Listing 4.1. Make it ask for a third string called website, which is the user's website used with the username and password. Ask the user to input the website after the user name and password are read.

Challenge 4.2: Code Listing 4.2 prints primes from 1 to 1000, at 10 per line. Without knowing the details of how this program works, can you change the program to print out the primes, from 1 to 50, and print two primes on a line instead of 10?

Challenge Answers

Challenge 4.0

Code Listing 4.3: Challenge 4.0

```java
import java.util.Scanner;
public class MainClass {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("What is your age?");
        String age = scanner.nextLine();
        System.out.println("You are " + age + " years old!");
    }
}
```

Challenge 4.1

Code Listing 4.4: Challenge 4.1

```java
import java.util.Scanner;
public class MainClass {
    public static void main(String[] args) {
        // Declare all variables.
        String userName;
        String userPassword;
        String website;
        Scanner scanner = new Scanner(System.in);

        // Request the user's name.
        System.out.print("What is your name? ");
        userName = scanner.nextLine();

        // Request the user's password.
        System.out.print("Ok, and what is your password? ");
        userPassword = scanner.nextLine();
        System.out.print("And what website is this the password for? ");
        website = scanner.nextLine();

        // Print out a message:
        System.out.println("Ok, gotta go. I have to quickly check " +
                website + ".");

        // Close the scanner to avoid memory leaks.
        if(scanner != null)
            scanner.close();
```

```
        }
}
```

Challenge 4.2

I have only included the relevant sections in the listing here, and I have highlighted the changes we need to make in yellow.

Code Listing 4.5: Challenge 4.2

```
int i = 2, count = 0, countPerLine = 2;

// Show a description of the program.
System.out.println("Prime numbers from 1 to 50 are as follows: ");
// Loop that finds the primes
while(i <= 50) {
      // If i is prime...
        if(IsPrime(i)) {
```

Chapter 5 Data Types and Variables

Memory

In Java, as with all computer programming languages, we can read and write data to various memory spaces. A memory space is simply an area in which we can save and retrieve information. One of the most useful memory spaces that we can read and write to is called system RAM. RAM stands for Random Access Memory. It is a type of volatile memory used for temporary storage (volatile means it is cleared when the system reboots).

RAM is nothing more than a long string of 0s and 1s. The 0s and 1s are organized into bytes of 8 bits each (a bit is short for binary digit—it is a single 0 or 1). See Figure 35 for an illustration of several bytes in RAM.
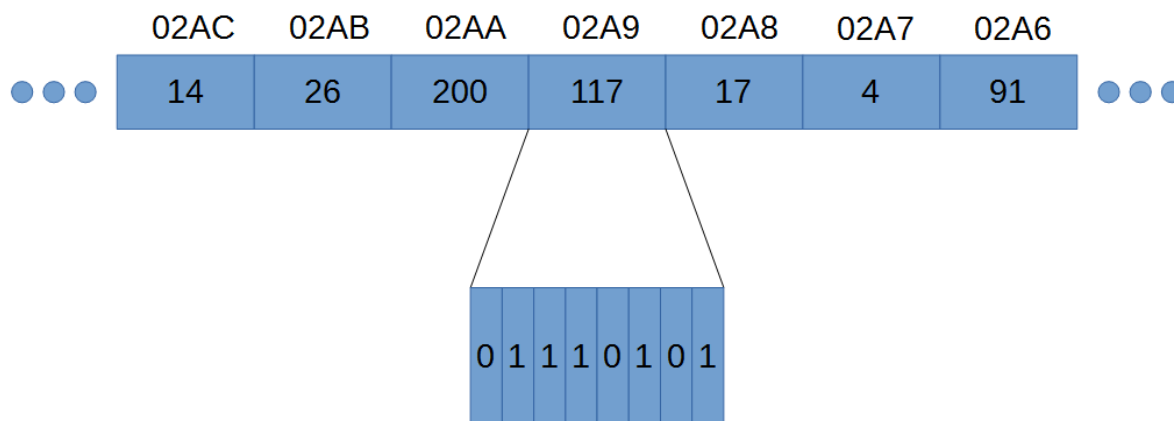
Figure 35: RAM

Figure 35 shows seven bytes in RAM. Each byte in RAM has an address written above the boxes (real addresses on modern computers have 16 hexadecimal digits instead of the four I've used in the illustration). The addresses are simply consecutive numbers counting up as we move through RAM from right to left. The addresses are written in hexadecimal (so we have 02A8 instead of 680—hexadecimal is not important at the moment).

📝 ***Note: In Java, we do not have access to the addresses in RAM. This is one of the biggest differences between Java and related languages C and C++. Accessing RAM directly with addresses can be dangerous—it is easy to accidentally address the wrong bytes and cause bugs in our programs. So, Java does not allow direct access to RAM.***

Along with an address, each byte in RAM also has a value (such as 14, 100, or 117 in Figure 35). As you can see, RAM is a collection of bytes, each with an address so that we can reference the bytes, and a value, so that we can read and write some information.

In Figure 35, the byte at address 02A9 is zoomed in, and we can see the individual bits that make up the number 117 as written out in the binary counting system. In binary, bits are read from left to right, the same way we read normal decimal numbers. So, the bit on the left is the most significant (as in the 6 in 684), and the bit on the right is the least significant (as in the 4 in 684).

Programming a computer is difficult when we reference RAM by addresses (and Java does not allow us direct access to bytes by their addresses). Addresses are simply long hexadecimal numbers, and they do not appear meaningful to humans.

As programmers, we need to manipulate and control hundreds, thousands, and even millions of bytes in RAM, and doing so with addresses is extremely difficult. Therefore, instead of referencing RAM by addresses, we name certain points in RAM and use the names instead of the addresses. These named points in RAM are called variables. A variable is literally a point that we have named and in which we can read and write in RAM. Variables often contain multiple bytes—for instance, an `int` variable in Java is built from four consecutive bytes and can be used to store larger numbers than a single byte can store.

Each variable in Java has a data type that is an indication as to what type of manipulations we will perform on the particular spot in RAM or which type of information the bits represent. For instance, we might wish to use a variable for floating-point computations such as 3.4+7.8, or we might wish to perform some operation with whole numbers (integers) such as 3+7.

Primitive data types

The most fundamental, basic building blocks of data types in Java are called the primitive data types (sometimes just primitives). They are used to build more complex data types such as classes and enumerations. When we declare variables in Java, we must state their data types. Java is a type-safe language, which means that once a data type is given to a variable, we cannot later change the type.

Table 1: Primitive Data Types

Name	Type	Size (Bytes)	Size (Bits)	Minimum	Maximum
byte	Integer	1	8	-128	127
short	Integer	2	16	-32768	23767
int	Integer	4	32	-2147483648	2147483647
long	Integer	8	64	-2^{63}	$2^{63}-1$
float	Float	4	32	-3.4×10^{38}	3.4×10^{38}
double	Float	8	64	-2.2×10^{308}	2.2×10^{308}
boolean	Boolean	1*	8*	false	true

Name	Type	Size (Bytes)	Size (Bits)	Minimum	Maximum
char	Character	2	16	Unicode	Unicode

* The size of boolean variables is dependent on the virtual machine that executes the code. Typically, the most conservative size of one byte is used.

Table 1 shows all eight of the primitive data types available in Java. The first column shows the keywords we use in Java when we declare variables (we will look at declaring variables shortly).

The Type column shows the fundamental data type of each primitive. There are only two data types: Integer and Floating Point. Integer data types are whole numbers—they can be negative and positive—but they cannot have any decimal part: e.g., 6, 2378, or -1772. Floating-point data types are used to represent values with some fractional part, such as 28.7 or -1772.1671.

The Boolean and Character types are actually integers, but there are various functions available in Java that treat these particular primitives differently than regular integers. For instance, when we print the value of a `boolean` variable, we will get `true` or `false` printed to the screen instead of 1 or 0 (even though the underlying data is literally an integer where 0 means `false` and anything else means `true`). And, when we print the value of a `char` variable, Java will print out the Unicode symbol that corresponds to the integer value of the `char` variable.

The third column shows the sizes in bytes for each of the primitive data types. Figure 36 shows an illustration of the sizes of each primitive. Each box represents one byte of RAM. The numbers show the most common byte order—Java is Little Endian, the lowest byte in a multiple-byte data type, on the right side. Floats and doubles use a specific format that is very different from integers. The format of floats and doubles in Java follows the IEEE 754 Standard, which is a scientific notation for representing approximations to fractions in binary.
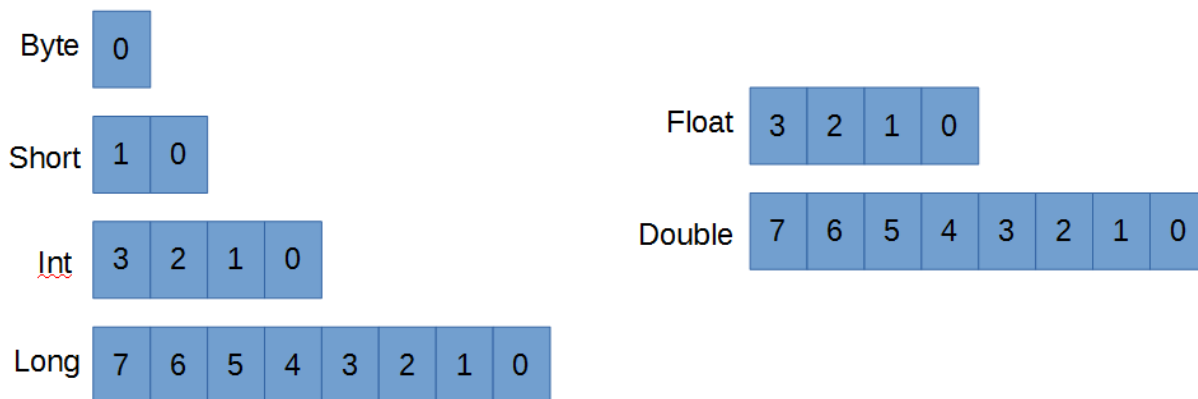

Figure 36: Primitive Data Type Sizes

The fourth column shows how many bits (binary digits) each primitive takes up in RAM. This will always be a multiple of eight because RAM is a series of bytes, and bytes are eight bits each. The exact workings of binary are not important at this point (we will look in some detail at binary when we look at the bitwise operators), but it is easy to represent any number in binary that you can represent in base 10. For instance, the number 219 in base 10 is the same as 11011011 in binary.

The final two columns show the available ranges of the primitives—the minimum and maximum values that variables of each primitive can store.

Programmers should become familiar with all eight of the primitive data types, but in practice we tend to use some types more than others. Most programmers will use `int` for all numerical, whole-number variables. The smaller integer data types, `byte` and `short`, are used to save space in RAM when the range of values being stored is known to fall within the much smaller range of the `byte` or `short`. Similarly, the `long` data type is not typically used unless the programmer has a specific reason to do so—for instance, if the programmer needs to store values greater than 2.15 billion.

Note: If you require more range than a `Long` can accommodate, there is a standard class called "BigInteger" that offers arbitrarily large integers. `BigInteger` is not a primitive, but instead a class built of primitives that allows computation with very large values (used in encryption algorithms, for example). This range of computation comes at a cost, and `BigInteger` computations are much slower for the JVM to perform than computations with `int` or `Long`. In order to use a BigInteger, we need to import `java.math.BigInteger` and declare a `BigInteger` object.

Programmers often use `double` as the default data type when they need to represent floating-point values. A `double` is twice as large as a float (64 bits for doubles, 32 for floats), which means that if you have a lot of variables (such as an array with one million elements), storing the elements as `float` will mean the array takes roughly four megabytes of RAM rather than eight. The downside to `float` is that it is far less accurate. The extra accuracy offered by doubles can be very important because every operation we perform on floating-point variables (`double` and `float`) has the potential to introduce and magnify rounding errors.

The boolean type has only two possible states—`true` and `false`. We use `boolean` variables when we know there are only two states to a variable, and the states need not necessarily be `true` and `false`. For instance, a `boolean` variable could represent a person's gender, a light switch being on or off, or the digits 0 and 1. And `boolean` values are commonly used to create conditional, or logical, expressions. We often use a logical expression without actually defining a variable, but behind the scenes a `boolean` variable is being created and evaluated for every logical expression that is executed (we will see many logical and conditional expressions in the section on control structures).

The `char` data type is used to store letters, digits, punctuation, and many other symbols (`char` represents characters using the Unicode encoding scheme). The `char` data type is really just a `short`, but the JVM treats the 16 bits differently. For example, if you set a `short` to 65 and print the results using a `println`, you will see 65. But if you use a `char` variable, set it to 65 and print the results with a `println` the program will print A. This is because the `println` function is designed to treat `char` data types as references to Unicode characters. The symbol in the Unicode table at position 65 is A, therefore `println` prints A to the screen instead of 65 when it sees that the data type is `char`.

`char` variable data types are strung together to form strings of characters. Strings are simply text, they are not a primitive type, and we will cover them in a moment.

Variables

A variable is a named, primitive data type. Each variable is a collection of bytes acting as one of the primitive data types from Table 1, and which we have given a name. We refer to variables by their name (instead of their addresses in RAM) and can read or change their values of RAM using the variable names as our references. A variable stores information, such as the number 3 or the name Jane Smith.

Variable declaration and definition

To declare a variable is to state that it exists. We give the variable a data type and a name—that is, an identifier. To define a variable is to specify a value for it. Variables must be declared and defined before they are used.

Initialization comes when we define a variable's value for the first time. A variable that is declared but not defined is called uninitialized, and the variable cannot be used in computation until it is initialized. In Java, we can declare and initialize a variable in a single line of code using the assignment operator "=" (see the section on operators for more information on the assignment operator).

Code Listing 5.0 shows various declarations and definitions of variables.

Code Listing 5.0: Declaring and Defining Variables

```java
public class MainClass {
    public static void main(String[] args) {
        int i = 100; // Declare and initialize an int.
        int b = 10, c = 5;// Declare and initialize two variables.
        boolean mb = false;// Declare and define a Boolean.

        char myLetter; // Declare a character.
        myLetter = 'h'; // Define/set the character to be 'h'.
        byte hp, bp, sp; // Declare 3 byte variables.
        hp = bp = sp = 0; // Set all 3 bytes to 0.
        short ss = 189; // Declare and define a short integer.
        long p = ss; // Declare and define a long with the value of ss.
        double myDouble = 100.0;// Declare and define a double.
        float f = 0.1f; // Declare and define a float.
    }
}
```

When declaring a variable, the data type comes first, followed by the name of the variable, and optionally followed by the initial setting for the variable. Variables can be defined on the same line as the declaration (such as the `int` variables a, b, and c in Code Listing 5.0) or on a separate line (such as the char variable `myLetter` in Code Listing 5.0). You can declare and define more than one variable of the same type by separating each variable with a comma (as shown by the variables b and c in Code Listing 5.0).

The numbers in Code Listing 5.0, such as `189` and `0.1f`, as well as the character `'h'` and the boolean value `false`, are called literals. We will examine the literals later.

You can also use mathematical and logical statements to initialize a variable. For example: `int someInteger = 100*9+1`. This will set the variable `someInteger` to `901`.

Tip: Name your variables so that their purposes are obvious. Carefully selecting names for variables can greatly increase the readability of code. In general, a programmer should name variables succinctly. Consider "s" as a variable to store an employee's salary vs. the more appropriately named "employeeSalary." Calling the variable "employeeSalary" is much more descriptive than the letter "s."

Note: As mentioned, many programmers use a technique called Camel Case to name identifiers when programming with Java (and other languages). The first word of a member method or variable's name is lowercase, and all subsequent words begin with an uppercase letter. For example, "myVariable" and "vehiclesPerHour." But class names begin with an uppercase letter—"MyClass" and "UserInformation."

Detour: strings

Before we move to describing literals, let's take a brief detour to examine a data type that is not a primitive. `String` is not a primitive, but it is so commonly used that it can be introduced with the other data types. A `String` is an array (see the section on arrays) or sequence of `char` variables, one after the other. A `String` can be used to represent text, numbers, or sequences of Unicode symbols (a `String` is literally a string of `char` variables). We surround text with double quotes to indicate that it is a `String`.

```
String str = "This is a string variable!";
```

We can define a `String` on multiple lines by using the + operator to connect more than one `String` together (Code Listing 5.1).

Code Listing 5.1: Multiple-Line String

```
String description = "Strings can " +

    "be defined on multiple " +

    "lines with the + operator!";
```

There is a very important distinction to make between a `String` and numerical data types such as `int` and `float`. A `String` is not interchangeable with a numerical data type, even if both have the same value. A `String` can be parsed into integers using a function call, `Integer.parseInt`, but it is not possible to assign a `String` to an integer in the normal way. The second line of Code Listing 5.2 is illegal:

Code Listing 5.2: Strings and Numerical Data Types are Not Interchangeable

```
String str = "100";

int j = 10 + str;
```

A `String` is an array of characters, and each character has a more or less arbitrary numerical value (it is the Unicode character set). If we add 10 to the character 7, we do not get 17, but 65! This is because the character 7 is not equal to the integer 7. The character 7 has a Unicode value of 55. For this reason, we are not able to convert a numerical string such as 1278 to the integer 1278 without calling a helper function such as `Integer.parseInt`.

Code Listing 5.3 shows parsing a `String` to an `int` and adding a character to an integer so that these operations will give the normal, expected results. We will cover calling functions, boxing/unboxing, and the `Integer` class later. Let it suffice to say that `Integer.parseInt(str)` will return the `int` 7, the numerical value of the characters of the `String`.

Code Listing 5.3: Parsing Strings and Characters

```java
public class MainClass {
    public static void main(String[] args) {
        // Set str to the string "7".
        String str = "7";

        // Set j to the integer 10 + "7" parsed to int.
        int j = 10 + Integer.parseInt(str);

        // Set c to the character '7'.
        char c = '7';

        // Set i to 10 + 7, notice we subtract the
        // character '0' from '7' to get the int 7
        // from the character '7' instead of parsing.

        int i = 10 + c - '0';

        // Both j and i are set to 17.
        System.out.println("j: " + j);
        System.out.println("i: " + i);
    }
}
```

Code Listing 5.4 shows an example of adding an integer to a `String`.

Code Listing 5.4: Adding Integers to Strings

```
int idNumber = 26771;

String desc = "The ID number is " + idNumber + ".";
```

In Code Listing 5.4, we use the concatenation operator, +, to place the value 26771 in the middle of a `String`. Java will create the value "The ID number is 26771" and store this in the `String desc`. It is important to note that the integer `idNumber` has been converted to a normal decimal string automatically.

Casting

To cast in Java means to change the value of a variable or literal to that of another primitive data type. To cast from one type to another, we use the primitive we want to cast to in brackets. Code Listing 5.5 shows some examples of casting numerical data types.

Code Listing 5.5: Casting

```
int i = (int) 7.8;        // Cast the double 7.8 to an integer, store in i

double d = (double) 89.7f;// Cast the float 89.7f to a double, store in d

short q = (short) (i * d);// Cast the double i*d to a short and store in q
```

Casting a floating-point value to an integer type results in truncation, so that 7.8 becomes 7 when stored in the integer i.

This example shows something else interesting. The expression (i*d) results in a `double`. When we perform an operation and mix integers and doubles, or floats, the integers will be temporarily changed to double or float. That is, integers are implicitly cast to double temporarily when performing mixed arithmetic.

Note: Each of the fundamental data types also has an object version with a similar name: Integer, Double, Short, etc. All are classes used to wrap the fundamental data types. The classes contain useful static methods that can be used for parsing strings to integers or doubles. Changing a fundamental data type to its class version (e.g., `int` to an Integer) is called Boxing. Changing a boxed Integer back into a primitive `int` is called Unboxing. For more information on Boxing and Unboxing, visit Oracle at https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html.

Literals

A literal is a value written in the code, such as 27, the character A, or the floating-point value 78.19f. We have seen many literals already. We often set the values of variables to literals or use literals in computations in order to produce some result, then store it in a variable.

Integer literals

Integer literals are simply whole numbers. Integers can be positive or negative. Negative integer literals have the minus sign to the left of the left-most digit (e.g., -27). For positive integer literals, the plus sign, +, is optional and almost never used (i.e. it would be more normal to use 78 rather than +78, but both are legal).

Code Listing 5.6: Integer Literals

```java
int i = 100;
int j = -239;
short p = 99;
byte q = -55;
long l = 67876123567651;

int h = 0x5DD2; // Hexadecimal
int w = 067; // Octal
int bin = 0b10011001; // Binary literal, Java SE 7 and above
```

Code Listing 5.6 shows a code snippet with several integer variables declared and defined using integer literals. The literals are on the right side of the assignment operator: =. A simple integer literal is an unadorned whole number such as the 100, -239, 99, and -55 in Code Listing 5.6. Such literals are interpreted by the JVM as being normal decimal numbers, and they are converted to binary behind the scenes.

Long literals have an "l" appended to the end (note that this is a lowercase L), as the line "**long** l = 67876123567651;" shows. You can use a regular int literal to set the value of a long integer, but only if the literal does not fall outside the range of an int. Java will automatically convert the int literal to a long literal to store in the variable when the variable is of type long. If the literal is too large for a standard int to hold, it must have the "l" suffix.

Typically, programmers use base 10 when specifying integer literals. Literals without extra prefixes are assumed to be in base 10 (such as 100, -239, and 99 in Code Listing 5.6) because this is the most common counting system. The final three literals illustrated in Code Listing 5.6 show how to define integer literals using the hexadecimal (base 16), octal (base 8), and binary (base 2) counting systems. Sometimes it is easier to use hexadecimal or octal because these counting systems match the binary system that the computer uses more closely than does base 10. In order to define a hexadecimal integer, you begin the literal with 0x. In order to define an octal literal, we use a leading zero. This can be quite confusing, so it pays to be very careful when reading literals in more advanced code—the number 067 is not 67 in decimal, it is 55.

The final example in Code Listing 5.6 is a binary literal. Binary literals are only provided in Java SE 7 (released in 2011) and above. They allow us to express an integer by directly specifying its bits. In order to use a binary literal, begin the literal with the `0b` prefix. The literal `0b10011001` means 153 in base 10. The binary string in Code Listing 5.6 has only eight bits, but it could consist of up to 32 bits. I have used this literal to set the value of the variable `bin`, the remaining bits on the left of the eight bit will be filled with 0. The entire value of the `bin` variable will be `00000000 00000000 00000000 10011001`.

Floating-point literals

A floating-point number is a number whose decimal point can move, such as `15.6` becoming `1.56`. This is fundamentally different from an integer, which does not have a decimal point. Floating-point literals have two primitive types—`float` and `double`. When using `float` literals, we place "f" on the end of the literal. You'll see that `double` literals are unadorned, but they should always have a decimal point (i.e. instead of `6`, we would write `6.0` in order to show that the 6 is not an integer but a `double`). Code Listing 5.7 shows some examples of floating-point literals being used to initialize `float` and `double` variables.

Code Listing 5.7: Floating-Point Literals

```
float y = 99.0f;
float q = -3.4512f;
double h = 99.0;
double g = -3.4512;
double p = 2.4e3;
double u = 2.4e-3;
```

The final two examples in Code Listing 5.7 use scientific notation. When we use `2.4e3`, what we mean is 2.4×10^3 (or 2.4 by 10 to the power of 3). Likewise, `2.4e-3` means 2.4×10^{-3} or 0.0024.

You must be aware that `float` and `double` types rarely store the values we actually specify. Although perhaps a little strange, there is no IEEE 754 `float` for the value 1/3, or for 3/5. The format simply does not include a representation for these simple fractions. When we set a `float` or `double` to 1/3, we are really setting it close to 1/3. See the section on floating-point arithmetic for more details, and always keep in mind that `float` and `double` types are merely approximations of the values we specify.

Boolean and character literals

Code Listing 5.8: Boolean and Character Literals

```
boolean b1 = true;
boolean b2 = false;
char m = 'h';
char g = 65; // 65 is 'A' in Unicode!
char c = 0b1000001; // 65 in binary, this is also the character 'A'.
```

Code Listing 5.8 shows some examples of `boolean` and `char` literals. There are only two `boolean` literals—`true` and `false`. Character literals are almost always surrounded by single quotes such as `'h'`, but it is possible to use an integer literal to set a character (look up the Unicode character chart at http://www.unicode-table.com/en/ for a table of the Unicode characters and their corresponding integer values, and note that the table is laid out in hexadecimal).

Note: As with the integer literals, it is perfectly valid to express the value of a `char` in hexadecimal, octal, or binary. But this is very rare, and, typically, if we need to dictate the value of a variable using numerical digits, we should use a short instead of a `char`.

String literals

`String` literals are surrounded by double quotes, as opposed to single quotes used for `char` literals. You can continue long string literals by splitting them onto separate lines using the "+" operator (this is called concatenation of `String` literals, as we will see in the following chapter). Code Listing 5.9 shows examples of `String` literals.

Code Listing 5.9: String Literals

```
String m = "This is a string literal";
String j = "If your string literal is too long, you can " +
        "use the + operator and continue on the next line!";
```

Detour: arrays

An array is a collection of items in which all items are of the same data type. For instance, a collection of 10 ints or a collection of 1,000 doubles. Arrays can be created to store collections of user types, too, such as classes and enumerations. (We do not cover `enum` in this book, but it is a special type of class designed to store a series of constants.) The elements of an array are numbered starting with 0. We can use the array's name along with an index in square brackets, [and], in order to specify to which element we wish to read or write.
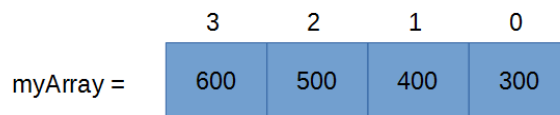
Figure 37: Array of Integers

Figure 37 shows an array of four integers called `myArray`. Each element in the array is represented by a box with two numbers. The number inside the box is the value of the element, and the number outside the box is the index of the element within the array. Notice that the first box (the one on the right) has an index of 0, not 1. And the final box (on the left) has an index of 3 instead of 4. This is extremely important, and it may take some getting used to—Java counts

from 0, not 1. So, if we create an array with 10 integers, the elements will be numbered from 0 to 9. There are still 10 items in the array—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, but they are not numbered in counting order.

Code Listing 5.10: Declaring and Defining an Array

```java
public class MainClass {
      public static void main(String[] args) {
            int[] myArray = new int[4];
            myArray[0] = 300;
            myArray[1] = 400;
            myArray[2] = 500;
            myArray[3] = 600;
            System.out.println("Element 0 is: " + myArray[0]);
            System.out.println("Element 3 is: " + myArray[10-7]);
      }
}
```

Code Listing 5.10 shows an example of how we might set up the array illustrated in Figure 37. The declaration of the array is `int[] myArray = new int[4];`. Notice the [] beside the data type. These brackets next to a data type declaration mean we do not want a single integer but rather an array to be declared. Then, after the name of the array, `myArray`, we use the assignment operator and the `new` keyword to specify how many elements the array should hold. The `new` keyword is also used to construct objects, as we will see. Code Listing 5.10 also shows how we set the individual elements of the array. We use the array's name, `myArray`, followed by the index of the element in square brackets [and].

When we want to read the values of the elements of an array, we use the square brackets again, as illustrated by the `System.out.println` calls in Code Listing 5.10. Note that in the second `println`, we use an expression in the square brackets, [10-7], instead of a simple integer. We can use any expression we like in the square brackets to index array elements, but the expression must be integer—we cannot reference the element at position 2.5 because all elements in an array have whole number indices (we will cover arithmetic expressions in some detail later). This means we are able to index array elements with variables, too, such as `myArray[i]`, in which `i` might be an integer variable.

We must be very careful to ensure that we never index outside the bounds of an array. The array in Code Listing 5.10 has four elements numbered 0, 1, 2, and 3. We cannot index array element `myArray[-15]` or `myArray[4]` or `myArray[2367]` because these elements do not exist. Indexing outside the bounds of an array is relatively safe in Java compared to languages such as C, but it will cause an exception to be thrown that can possibly crash the application.

Code Listing 5.11: Array Initialization Syntax

```java
public class MainClass {
      public static void main(String[] args) {
            int[] myArray = {
                        300,
                        400,
```

```
                        500,
                        600
            };
            System.out.println("Element 0 is: " + myArray[0]);
            System.out.println("Element 3 is: " + myArray[10-123]);
        }
}
```

Code Listing 5.11 shows the same array as Code Listing 5.10, but here we use a different syntax to set the values of the array. This is called array initialization. We follow the name of the array with the assignment operator, then a comma-separated list of elements between curly braces { and }. When we initialize an array using this syntax, we do not need to specify the size of the array because the compiler can infer the size from the number of elements we use in the curly braces.

Multidimensional arrays

A multidimensional array is a collection of items that are accessed using more than one index. For example, a grid of 10x50 integers or a 3-D box filled with 7x9x10 Boolean values. The basic arrays we have examined use one index in order to access any item, but grids and blocks of variables can be collected into multidimensional arrays. The number of dimensions in an array specifies how many indices we need in order to exactly reference any one element. For instance, a three-dimensional array requires three indices, one for each dimension, while a 2-D array requires two indices, and a 12-dimensional array requires 12 indices, etc.

Note: As the number of dimensions of an array increases, the amount of memory required to store the array increases dramatically. For instance, a 2-D array of bytes with dimensions 5x7 requires 5*7 bytes to store. However, a 3-D array with dimensions 5x7x8 requires 5*7*8 bytes to store. For this reason, we don't often see multidimensional arrays with dimensions numbering three or four.

Tip: Do not try to envisage what higher dimensional arrays look like (4-D, 5-D, etc.). Multidimensional arrays can be confusing to imagine. We live in a 3-D world, so it's easy to imagine a 2-D grid or a 3-D box full of array elements. However, it is difficult to envisage what a 4-D or a 5-D array of elements looks like. It really doesn't matter what the array would look like, though. All we care about is reading and writing the elements. It is no more difficult to read/write an element in a 2-D array than it is to read/write an element in a 12-D array. With 2-D we would use two indices, and for 12-D we would use 12 indices.

Code Listing 5.12: Declaring and Defining a 2-D Array

```
public class MainClass {
     public static void main(String[] args) {
            float[][] array2d = new float[7][5];
```

```java
            array2d[0][0] = 50.5f;
            array2d[4][4] = 60.5f;
            array2d[3][1] = 10.3f;
            System.out.println("Element (3, 1): " + array2d[3][1]);
            System.out.println("Element (4, 0): " + array2d[4][0]);
        }
}
```

Code Listing 5.12 shows how to declare a 2-D array of `float` types. We use the data type `float[][]` to signify that we wish to declare an array with two indices per element. The name of the array, `array2d`, is followed by the assignment operator and the `new` keyword. Then we supply the number of elements in each dimension of the array, 7 and 5. This means we are creating a grid of 7x5 floating-point values. Figure 38 shows two ways we might illustrate this array.
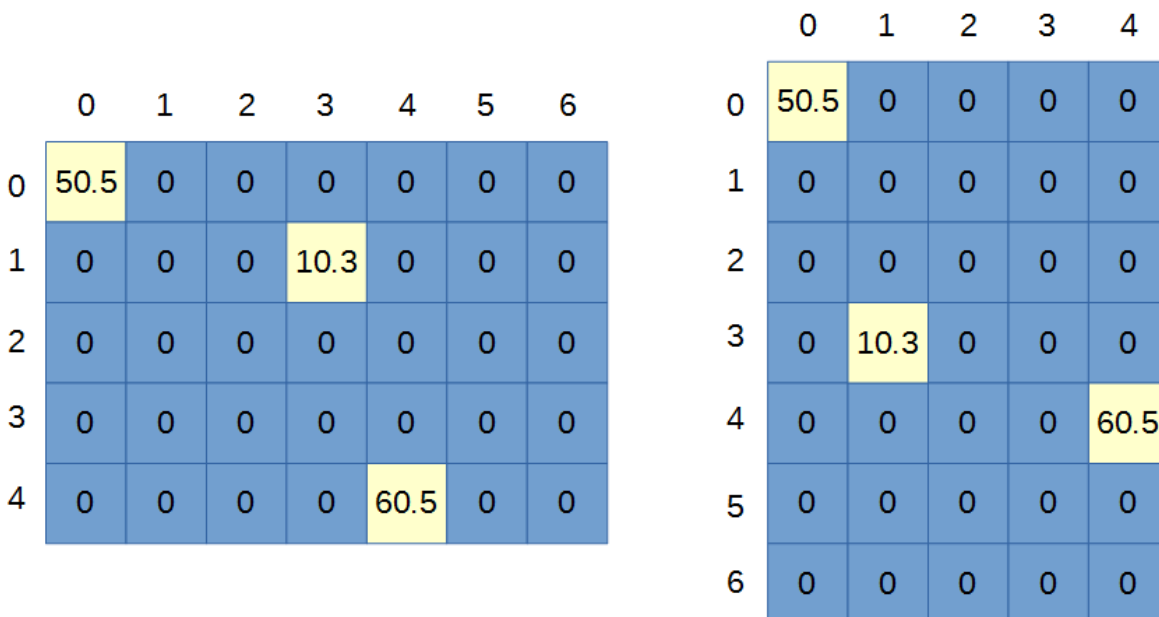
Figure 38: 2-D Array

Figure 38 shows two depictions of the same array defined in Code Listing 5.12. With a 2-D array, we are essentially representing rows and columns, but the orientation is up to the programmer. The trick to properly manipulating 2-D arrays is to envisage the array in the same orientation each time. The important thing is not the orientation of the grid but the fact that it has 7x5 elements. We can view this grid as having seven columns and five rows (the illustration on the left in Figure 38), or we can view the grid as having seven rows and five columns (as depicted in the illustration on the right of Figure 38). It does not matter which method (and indeed there are more orientations for a 2-D array than these two!) we use to illustrate a 2-D array, the important fact is that there are seven indices in the first index of the array and five in the second.

Let's next look at for loops. My objective here is to demonstrate how to correctly traverse (run through every element of) a 2-D array, rather than to explain the syntax of the loop itself.

Code Listing 5.13: Using Nested For Loops to Traverse a 2-D Array

```java
public class MainClass {
      public static void main(String[] args) {

            float[][] array2d = new float[7][5];

            for(int y = 0; y < 5; y++) {
                  for(int x = 0; x < 7; x++) {
                        array2d[x][y] = x * y;
                  }
            }
      }
}
```

Code Listing 5.13 shows how to traverse an entire 2-D array. It uses nested for loops, the outer for loop counts from 0 up to 5 (i.e. 0, 1, 2, 3, 4). Each time this counter (y in the code) is incremented, another counter, x, runs from 0 up to 7 (i.e. 0, 1, 2, 3, 4, 5, and 6). Thus, the inner body of these two nested for loops will run 7x5 times, and each time, the x and y iterators will be able to access a different element of the 7x5 array. In the body of the loop, we set the elements of the array as the product of their indices. Figure 39 shows the result in the pattern in the array after these loops are executed.

	0	1	2	3	4	5	6
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	2.0	3.0	4.0	5.0	6.0
2	0.0	2.0	4.0	6.0	8.0	10.0	12.0
3	0.0	3.0	6.0	9.0	12.0	15.0	18.0
4	0.0	4.0	8.0	12.0	16.0	20.0	24.0

Figure 39: 2-D Array after For Loops

2-D arrays are usually one of the most daunting aspects of Java for new programmers. However, I hope to illustrate that there is nothing difficult about these data structures at all—it's only a matter of getting the indices in the right order every time. The final example in this chapter shows how to declare and traverse a 4-D array, setting every element in the entire array

to 1729 with a few nested for loops (again, we will cover for loops in more detail in the control structures section, and it would be a very good idea to read that chapter and return to this example).

Code Listing 5.14: Creating and Traversing a 4-D Array

```java
public class MainClass {
    public static void main(String[] args) {
        // Declare 4-D array.
        int[][][][] arr4D = new int[3][4][5][6];

        // Set all the elements to 1729.
        for(int a = 0; a < 3; a++) {
            for(int b = 0; b < 4; b++) {
                for(int c = 0; c < 5; c++) {
                    for(int d = 0; d < 6; d++) {
                        arr4D[a][b][c][d] = 1729;
                    }
                }
            }
        }
    }
}
```

It is impossible to imagine what 4-D shapes look like. We cannot draw the array from Code Listing 5.14 on a piece of paper. And yet, manipulating and traversing the shape is as easy as 3.14159. All that matters is that the array has four dimensions with sizes 3, 4, 5, and 6, respectively. And, when we want to traverse the entire array, we can nest four for loops, each counting up to 3, 4, 5, and 6. We could set a single element with a statement such as `arr4D[1][3][2][5] = 87539319;`. But we must be careful that none of our indices goes outside the bounds for the particular dimension it references.

Also, the order in which we nest the for loops does not matter. The a counter happens to be on the outside loop in Code Listing 5.14, but we might have placed this for loop as the innermost one. The only important thing is that the counter is counting 0, 1, and 2, and we are using it to access a dimension that has elements indexed 0, 1, and 2.

Chapter 6 Operators and Expressions

Computers can add, subtract, multiply, and divide numbers extremely quickly. But basic numerical arithmetic is only one of the types of computation that computers do well. In this section, we will look at various methods for computing results and calculating other values for variables by using operators and expressions.

We will also look at some very important differences between the way computers perform arithmetic and the way that arithmetic works in mathematics.

Operators and operands

Operators are symbols used to indicate some specific action between one or two operands (in Java, the operators all take one or two operands). The arithmetic operators are actions such as addition, subtraction, multiplication, and division. Each operator has a specific symbol, such as "+" for addition and "*" for multiplication.

Operands are the parameters or the data for the operators. Each operator requires some predefined number of operands. For instance, addition requires two numbers, such as 5+6. Operands can be variables, objects, or literals (depending on exactly what the operator is). Operands specify the data that an operator operates on. The JVM reads the operands and the operator, it performs some bit manipulations based on the specifications of the operator, and this results in some value—for example, 5+6 results in the value 11.

Operators and operands are combined together with brackets "(" and ")" to form expressions. Expressions are similar to basic arithmetic, but in Java we have a lot more flexibility than basic arithmetic. We can form arithmetic, logical, Boolean, and string expressions.

Each operator has a predefined precedence that dictates the order in which the operators should be executed when they are combined in an expression, and brackets are used to override the normal precedence rules of the operators.

The general system we use in arithmetic and Java programming is called Infix Notation. In this system, operators with a higher precedence are performed first, followed by the operators with a lower precedence. For instance, 4+5*3 is an expression equaling 19. The precedence of the multiplication operator is higher than the precedence for the addition operator, so we perform 5*3 first, then add the 4, which results in 19. In Infix Notation, any operation surrounded by brackets occurs first, regardless of the precedence of the operators. So, we could write (4+5)*3, which has the same operators as before, but because of the brackets, the 4+5 is performed first and the final result is 27.

Arithmetic operators

Table 2: Arithmetic Operators

Name	Symbol	Operands	Precedence
Addition	+	2	1
Subtraction	-	2	1
Postfix Increment	i++	1	1
Postfix Decrement	i--	1	1
Multiplication	*	2	2
Division	/	2	2
Modulus	%	2	2
Unary Negative	-i	1	3
Unary Positive	+i	1	3
Prefix Increment	++i	1	4
Prefix Decrement	--i	1	4

Table 2 shows the arithmetic operators available in Java. Many will be familiar from regular arithmetic, but some require more explanation. First, I have included a dummy variable name `i` for the operators that require a variable in a certain position (e.g., the Postfix Decrement uses two minus symbols to the right of a variable `i`, like so: `i--`).

Table 2 also shows the relative precedence of the arithmetic operators. This is not the true precedence that the compiler uses when it parses expressions, but merely relative precedence so that you get an idea for which operations will occur first. You should always use brackets to directly specify the order of operations in complex expressions.

Addition, subtraction, multiplication, and division behave in similar ways to their arithmetic counterparts. But there is an important distinction between Java's operators and their arithmetic counter parts—all Java arithmetic is finite. This leads to some important points that are described in the sections on Integer Arithmetic and Rounding Error.

Postfix and Prefix increment and decrement take only a single parameter each. To increment is to add one, to decrement is subtract one. These operators can only be used with variables—you cannot increment a literal. The operators add one and subtract one from the value of the variable, and they set the variable accordingly (They are actually assignment operators, but they

are generally introduced with the arithmetic operators.). For instance, if j is an integer with 5 as the value, then j++ will cause j to become 6. Postfix and Prefix increment differ only in precedence. Postfix increment occurs before other operations because it has a high precedence, and prefix increment occurs after because it has a low precedence. Code Listing 6.0 illustrates the difference between Postfix and Prefix.

Code Listing 6.0: Postfix vs. Prefix Increment

```
int j = 10;
int b = ++j;        // j increment to 11.
                    // b is set to 11.


int p = 10;
int g = p++;        // b is set to 10.
                    // p increments to 11.
```

Unary negative and unary positive differ from their standard mathematical definitions. In this context, unary simply means "single operand" and binary means "two operands." Unary negative is the minus symbol when used beside a single number or variable, e.g., -9. In regular mathematics, there is no clear distinction between the precedence of unary negative -7 and binary negative 5-6, which can be read as a shorthand for 5+-6. In programming languages, the two are often implemented with the unary versions having a greater precedence than the binary versions. This is because programming languages such as Java have the concept of a negative number instead of applying a negative operator with some predefined precedence. When the compiler reads -7, it does not take 7 and subtract it from 0. It reads the value as a negative integer.

Also worth mentioning is the modulus operator: %. Modulus returns the remainder after division. For example, 67%5 is equal to 2, because 2 is left over when we divide 67 by 5. Modulus is a very important operator. If a variable j is evenly divisible by another k, then j%k=0. That is, if j divides k, then j%k leaves no remainder.

Code Listing 6.1 shows some examples of expressions using the arithmetic operators. I have included comments describing the values of the variables as the lines are executed.

Code Listing 6.1: Arithmetic Operators

```
public class MainClass {
    public static void main(String[] args) {
        int j = 190;            // j gets 190
        j++;                    // j gets 191
        j = j - 89;             // j gets 102
        int b = 78 * j;         // b gets 7956
        int c = b + (9*j) / 12; // c gets 8032
        b--;                    // b gets 7955
        c = c % b + (j / 2);    // c gets 128
        c = --b;                // c and b get 7954
        int h = (j + b) % 210;  // h gets 76
    }
```

```
}
```

Integer arithmetic: overflow

In Java, the integers do not have an infinite size. They wrap around when the results are too great in magnitude to fit into the designated variable. This is a very important difference between the arithmetic that computers use and the mathematics of the real world. And it can lead to some strange results if we are not careful.

Code Listing 6.2: Overflow Example

```java
public class MainClass {
    public static void main(String[] args) {

        // Causing overflow of a byte:
        byte myByte = 127;
        myByte++; // Results in -128
        System.out.println("Byte says 127+1=" + myByte);
        myByte = -128;
        myByte--; // Results in 127
        System.out.println("Byte says -128-1=" + myByte);

        // Causing overflow of a short.
        short myShort = 32767;  // This is the maximum for short!
        System.out.println("Short is " + myShort + " before ++.");
        myShort++;
        System.out.println("Short is " + myShort + " after ++.");
        myShort--;
        System.out.println("Short is " + myShort + " after --.");
    }
}
```

In Code Listing 6.2, we set the variable `myByte` to 127, which is the largest positive value that a byte can accommodate. In the very next line, we increment the variable using the ++ operator. Incrementing a variable set to 127 will normally result in the answer 128. But the value 128 is outside the range of a byte, which is from -128 to 127. Therefore, the result will actually wrap around, and what we get is 127+1=-128. Unless we are keenly aware of this wraparound behavior, it can quickly lead to bugs in our code as we count up in positive integers and suddenly pop out the other side with a very large negative value.

Similarly, on the next line we set the value of `myByte` to -128, and we then try to subtract 1 from this variable. Again, the result would normally be -129, but -129 does not fall within the range of the byte and the result will wrap around to positive 127.

The output from Code Listing 6.2 is as follows:

```
Byte says 127+1=-128
```

```
Byte says -128-1=127
Short is 32767 before ++.
Short is -32768 after ++.
Short is 32767 after --.
```

The range of byte and short is very limited, and this overflow can be easily illustrated. But we must remember that this wrapping around occurs when we use any integer type, even long. The range of an int is far greater than a byte, but it is not infinite.

Integer arithmetic: truncation

In addition to overflow, division of two integers will always round the result toward zero before storing. This may seem a poor choice at first, but it is a more useful way to perform arithmetic than regular rounding (i.e. rounding to the nearest whole number). The basic rounding rule is to chop off any digits right of the decimal point, which means 6.7 becomes 6, and 89.23678 becomes 89, and -2773.223 becomes -2773. This is a type of rounding called truncation.

Code Listing 6.3: Integer Rounding

```
int a = 24/5;      // Results in 4
int b = -24/5;     // Results in -4
```

Code Listing 6.3 illustrates the division of 24 and -24 by 5. On the first line, we divide 24 by 5 and store the result in the variable a. The result of 24/5 is actually 4.8—or 4 and 4/5. But because integers are whole numbers, the 4/5 is discarded. Notice the 4.8 is not rounded up to 5, but toward 0. The digits right of the decimal point are simply chopped off.

Likewise, on the second line of Code Listing 6.3, we see the division of -24 by 5 results in -4. Normally, the result would be -4.8, but again, integers can only store whole numbers and the digits to the left of the decimal point are chopped off (or truncated).

Truncation is particularly useful when dividing integers because we have the modulus operator to use in conjunction with the division. If we want to know exactly what 24/5 is, we cannot use floating-point numbers because (as we will see in a moment) floating point can only represent a handful of rational numbers accurately, and 4.8 is not one of them. But we can use division and modulus to create a fraction. Code Listing 6.4 shows a short example of how we can maintain the exact results of an integer division.

Code Listing 6.4: Integer Arithmetic and Fractions

```
public class MainClass {
    public static void main(String[] args) {
        int numerator = 24;
        int denominator = 5;

        int wholeResult = numerator / denominator;
        int fractionalResult = numerator % denominator;
```

```
            System.out.println("The result is " + wholeResult + " and " +
                    fractionalResult + "/" + denominator);
        }
}
```

Floating-point arithmetic: rounding error

Floating-point variables (`double` and `float`) can seem limitless because they have a very large range, and they can seemingly represent very precise fractional values. But floating-point variables are as limited as integers, and, as with integers, you should be aware of several caveats regarding floating-point arithmetic. The most important thing to keep in mind when using floating-point arithmetic is this—the more operations performed on floating-point values, the more error there will be in the result.

Floating-point variables can store only a specific set of rational results. We might assume that any real number can be stored in a `float` or `double`, but this is not true. The only numbers a `float` or `double` can store are sums of perfect powers of two and the number zero (`0.0`). Numbers such as 256, ¼, or 4096.1875 are all stored perfectly with no rounding in floating-point variables. Other seemingly simple numbers, such as 1/5 or 0.3333…, are rounded to a near approximation of their true value. This means that the more times we compute with numbers such as 1/5 and 1/3, the more error is introduced into the final result.

Also keep in mind that floating-point variables (`float` and `double`) are limited in precision in another way. The higher the floating-point numbers, the less precision we have in computing exact results. Integers with up to six digits are all represented exactly using a `float`, such as 18.0f, 27837.0, etc. But beyond six digits, only every second integer has a representable value. Floating point uses a specification called IEEE 754 (https://en.wikipedia.org/wiki/IEEE_floating_point) that exchanges accuracy for range. All integers represent a number exactly, but high in the floating-point variables there are gaps and some integers cannot be stored.

All of this means that every computation we perform with floating-point variables potentially introduces and/or magnifies rounding errors. And floating-point variables typically only store approximations of real numbers—they almost never store a fraction exactly.

Code Listing 6.5: Rounding Error in Floating-Point Arithmetic

```
public class MainClass {
    public static void main(String[] args) {
        float j = 901.0f/13.0f;
        float q = (53.0f / 13.0f)*17.0f;

        System.out.println("J: " + j + " Q:" + q);

        if(j != q)
            System.out.println("The two are not equal!");
    }
```

```
        }
```

Code Listing 6.5 illustrates two methods for computing 901/13 using floats. The listing uses an `if` statement, which we will cover when we look at control structures. Here the point is that both floats `j` and `q` should theoretically have exactly the same value. With some basic arithmetic, we could compute the same answer from the expression we used to set `j` and `q`:

901/13 = 69 and 4/13

(53/13)*17 = 69 and 4/13

However, there is no way to represent 4/13 as a finite sum of powers of two, therefore the computer will introduce some rounding errors in both of the above computations when using floats. This means that the two expressions will not result in exactly the same answer in Java, and the line "The two are not exactly equal!" will be printed to the console. The second result is likely to be less accurate than the first because it uses more operations, but neither result is correct. This is very important to remember when we are comparing floats, because it means we often need some small degree of error.

Code Listing 6.6: Comparing Floats with Error

```java
public class MainClass {
      public static void main(String[] args) {
            float j = 901.0f/13.0f;
            float q = (53.0f / 13.0f)*17.0f;

            System.out.println("J: " + j + " Q:" + q);

            if(Math.abs(j - q) > 0.001f)
                  System.out.println("The two are not equal!");
      }
}
```

Code Listing 6.6 illustrates the use of `Math.abs` method to allow a small margin of error between the two floating-point results. If the difference between the two floating-point values is less than 0.001, we assume they are "trying" to represent the same result. The function `Math.abs` takes a single numerical argument and returns the absolute value (i.e. if the argument is positive, then it is returned unchanged, and if it is negative, then it is changed to positive).

Tip: If you need to perform a common mathematical function, there is a good chance that the function is included in Java's Math library. Code Listing 6.6 shows how to call a function from the Math library by calling Math.abs and passing a parameter in brackets. The Math library contains trigonometric functions such as Math.sin and Math.cos, logarithm functions, min, max sqrt, and many others. It also contains very accurate constants such as Math.PI, which is double constant for the value of Pi, and Math.E, which is another double constant for the value e.

Assignment operators

The assignment operators are used to alter or set the values of our variables. The most common assignment operator is the assignment operator itself, which is represented in Java by a single equals sign, "=". This is very different from the double equal sign, "==", which is a conditional operator and which asks if two values are equal. If we want to set a variable called `someVariable` to the integer 78, we would use "`someVariable = 78;`".

After the equals operator, the remaining assignment operators (+=, -=, /=, *=, %=, &=, !, =, and ^=) are provided for convenience. They are sometimes called compound assignment operators, and they are a shorthand for a particularly common form of expression. When we have a variable, we often need to perform some operation on it and store the resulting value back into the same variable. This does not make sense mathematically, but it is perfectly normal in a programming language. Here is an example of a compound assignment operator:

```
Var1 += 1;
```

The above statement expands to mean:

```
Var1 = Var1 + 1;
```

So, the value of the variable `Var1` has 1 added to it. If `Var1` begins with the value of 25, it will be set to 26 after this line of code executes. The other assignment operators are similar only if the operation is specific to each operator. Instead of adding, `-=` subtracts from the variable and `^=` perform bitwise exclusive OR, etc. Code Listing 6.7 shows some examples of compound assignment operators.

Code Listing 6.7: Compound Operators

```
Var1 -= 7; // Means Var1 = Var1 – 7;

Var1 ^= 7; // Means Var1 = Var1 ^ 7;

Var1 /= 7; // Means Var1 = Var1 / 7;
```

We will look at the exact meaning of each of the unadorned versions of these operators (i.e. ^ as opposed to the assignment version ^=). We can use the compound assignment operators with expressions, too:

```
Var1 += 5 + (2* Var2);
```

The above line of code will expand to the `Var1 = Var1 + 5 + (2* Var2);`. If `Var1` is 10 and `Var2` is 5, then `Var1` will become 10+5+(2*5) or 25.

Detour: binary numbers

Before we look at the bitwise operators, we will take a brief detour and look at the way computers store integers. Computers perform all computation on collections of 0s and 1s. The smallest amount of memory in the computer is a bit that can be set to either 0 or 1 and that constitutes a single digit in binary. Variables, such as `int` and `byte` (even `double` and `float`) are made up of a collection of bits in binary.

Byte containing the decimal number 50 in binary
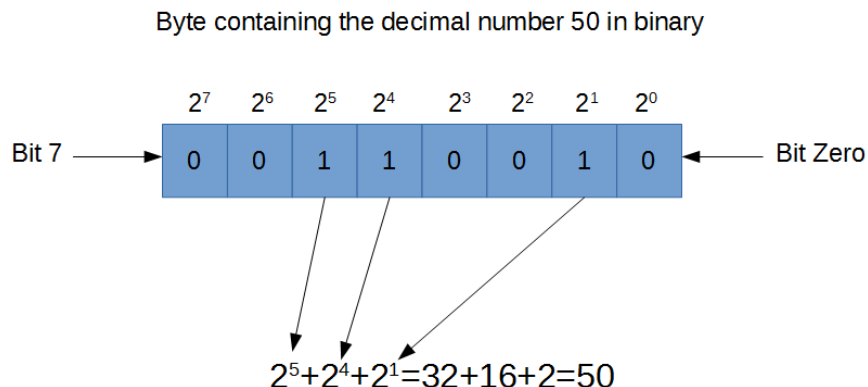
Figure 40: Byte Containing 50

Figure 40 illustrates a byte containing the decimal integer 50. The bits are blue boxes, and each bit is set to either 1 or 0 (the values inside the blue boxes). Each bit has an index. Bit 7, on the left, is the most significant bit and bit zero, on the right, is the least significant bit. This is exactly the same as with decimals—if we have a decimal number such as 345, the 3 on the left is more significant than the 5 on the right.

In order to work out the exact numerical value of a binary number, you must sum each power of 2 that corresponds to a 1 in the bits. In Figure 40, there is a 1 in the 2^5 position, another in the 2^4 position, and another in the 2^1 position. Therefore, this bit sequence represents 50, which is the result of adding the powers together: $2^5+2^4+2^1=50$. Notice that this is the same in decimal except that the base of the powers in decimal is 10 rather than 2.

Negative numbers are slightly more complicated than positives. Java uses twos-complement (2's complement). This means that if the left-most bit is set to 1, the number is negative. But the value of the negative number is the bitwise complement plus 1—e.g., the bit sequence 10000011 in a byte would normally be $2^7+2^1+2^0$, but because bit 7 is the sign bit in a byte, and because it is set to 1, the actual value this sequence represents is the complement +1. The complement is the same string with every bit flipped (see the ~ operator below). Therefore, 10000011 actually means 01111100+1=01111101, or -125 in decimal. The left-most bit in each of the integer data types is called the sign bit for this reason. For `byte`, this left-most bit is bit 7, and in `short` it's bit 15. In `int`, it is bit 31, and in `long` it is bit 63.

Bitwise operators

Several operators in Java are designed to directly manipulate the bits of our variables. These operators are often grouped together and called the bitwise operators. The bitwise operators are

borrowed from a logical mathematics called Boolean algebra (which is also where we get the keyword "Boolean" for our true/false variables and the boolean operators that we will see shortly).

Boolean algebra consists of variables and operators, as in regular algebra, and we can use brackets to override the normal precedence of the operators, too. In Boolean algebra, the variables can only be true or false, 1 or 0—there are no other values. In many ways, Boolean algebra is much simpler than regular arithmetic, but it is easy to underestimate the power of this apparently simple set of logical rules. Boolean algebra can be used to compute anything that is computable. In fact, modern computers are nothing but billions of transistors all connected together and performing basic Boolean algebra.

There are only three basic operators in Boolean algebra—AND, OR, and NOT. In addition to these three, most programming languages also provide a fourth operator—XOR, which is short for Exclusive OR.

In Java, we use the symbols & for Boolean AND, | for Boolean OR, ~ for Boolean NOT (or the bitwise complement), and ^ for XOR.

AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

NOT

A	Out
0	1
1	0

Figure 41: Truth Tables for Boolean Operators

Figure 41 shows the truth tables for the four bitwise operators available in Java. These tables are also the truth tables for the Boolean operators that we will look at shortly. Truth tables list the possible input states, and next to each state they list the result returned by the operator.

For instance, if we have the operator AND, and our two inputs (A and B) are set to 1 and 0, we can read along the third row of the AND truth table (which corresponds to A being 1 and B being 0) and see that the Out column says 0. This means 1 AND 0=0. Likewise, if we need to know what A^B is when A is 0 and B is 1, we can look along the second row of the XOR truth table and see that it will return 1 as the result.

In real Boolean algebra, the operations take only single bit operands, so we can compute the result from "True AND False" or "False OR False." However, computers do not typically operate on single bits because this would be too slow. Instead, computers group bits together into strings to form a byte, short, int, etc. A byte in Java is eight bits wide. This means any binary

value from 00000000 to 11111111 will fit into the eight bits of a byte. When we perform a bitwise operation between two variables, we actually perform the same Boolean operation eight times—once between each of the corresponding bits in two variables.

For instance, imagine we have two byte variables, a and b, that are set to 203 (assume the byte is unsigned, even though this would typically be read as a negative number using 2's complement) and 166, respectively. In binary, the numbers 203 and 166 are represented by the bit patterns 11001011 and 10100110. If we perform the bitwise operations between these variables and store the results in a third variable, c, Code Listing 6.8 shows what will happen.

Code Listing 6.8: Bitwise Operations

```
c = a & b;  // c will be set to 10000010

c = a | b;  // c will be set to 11101111

c = a ^ b;  // c will be set to 01101101
```

Note that whatever operation is being performed (&, |, or ^) the first bit on the right side of the resulting c value is the result of the operation performed between the right-most bits of the two input operands. And, as a final example, if we perform the bitwise complement of the a variable, we are simply flipping all of the 1s to 0s and all of the 0s to 1s:

```
c = ~a;// c will be set to 00110100, which is the complement of a
```

Shift operators

In addition to the bitwise operations based on Boolean algebra, we can also perform shifts in Java. The shifts are considered to be bitwise operators because they give us direct access to the bits of the variables. There are three shifts available in Java—shift left (the operator symbol is <<), shift right (the operator symbol is >>), and shift right with zero fill (the operator is >>>).

The shifts each take two operands. They shift the bits in the first operand to the left or right (depending on the operator) by the amount specified in the second operand. For instance, with a variable called a, we can shift the bits one space to the right by using the statement: a = a << 1;.

The effect of shifting the bits of integer to the left is the same as multiplying the value of the integer by the corresponding power of two. For instance, if we take the bit pattern three and we store it in a byte variable called a:

```
a = 3 << 4;// multiply 3 by 2 to the power 4, or 16, and a will become
00110000.
```

Likewise, shifting a variable to the right by some amount is the same as dividing the variable a power of two:

```
a = 48 >> 4;// divide 48 by 2 to the power 4, or 16, and a will become
00000011.
```

For the computer, shifting is much easier than division and multiplication, so it is used to optimize code (i.e. to make it run faster).

When we shift right, as bits shift out on the right side, new bits must enter on the left. When using the regular right shift (>>), the bits which enter on the left will be copies of the sign bit (i.e. the leftmost bit will be maintained). This enables us to divide negative numbers using the shifting technique described above and to maintain a negative answer. Whereas the shift right and zero fill operator (>>>) will shift in 0s on the left side. The shift right and zero fill operator is useful when we do not want Java to treat our numbers as being signed.

Relational operators

The relational operators allow us to compare the values of numerical variables against other variables or literal values. For instance, we might test if the integer variable `A` has a value less than `100`. Or we could test if the variables `B` and `C` are equal to the statement `B==C`.

The relational operators take two operands each. They compare the operands and return a `boolean` result of `true` or `false`. The relational operators are most commonly used in conditional statements such as if statements and loops (which will be covered in the Control Structures chapter). We can string together many relational operators with parameters using the logical operators to form complex expressions (see the next section for a description of the logical operators).

Table 3: Relational Operators

Operator	Name	Example
==	Equal to	A==B; // Is A equal to B?
!=	Not equal to	A!=B; // Is A different from B?
<	Less than	A<B; // Is A less than B?
>	Greater than	A>B; // Is A greater than B?
<=	Less than or equal to	A<=B; // Is A less than or equal to B?
>=	Greater than or equal to	A>=B; // Is A greater than or equal to B?

Table 3 shows the relational operators available in Java. Notice that the double equals sign, `==`, is the relational equal to operator. It asks if two variables have the same value. Whereas the single equals sign, `=`, is the assignment operator. It sets the value of the first operand to that of the second. Code Listing 6.9 shows some examples of how we can set `boolean` variables based on the results from logical. When we use relational operators in control structures, they generally do not set `boolean` values, but they execute code blocks when the results evaluate to true.

```
int var1 = 100, var2 = 30;

boolean ans1 = var1 == var2; // ans1 becomes false, 100 does not equal 30

boolean ans2 = var1 > var2; // ans2 becomes true, 100 is greater than 30

boolean ans3 = var2 <= var1; // ans3 becomes true, 30 is less than or equal
to 100

boolean ans4 = var2 != var1; // ans4 becomes true, 30 is not equal 100



int v1 = 100, v2 = 30, v3 = 200;

boolean ans1 = (v1 < v2) || (v3 > v1);    // Using logical OR
```

Logical operators

The logical operators (sometimes called Boolean operators) work with `boolean` arguments. They are used to form logical statements. Each logical operator takes two operands, and both operands are `boolean`. We usually use logical operators in conjunction with if statements, loops, and other circumstances in which conditions are necessary.

The three logical operators are AND, represented by the symbol &&; OR, represented by the symbol ||; and NOT, represented by the exclamation point symbol !. The logical operators are used to string together multiple simple `boolean` conditions in order to create more complex expressions. Each of the logical operators works in exactly the same way as the Boolean truth tables for AND, OR, and NOT that we looked at for the bitwise operators. The only difference between the logical operators and the bitwise operators is that the logical operators are strictly meant for `boolean` parameters, whereas the bitwise operators are designed to work with all 32 bits of `int`, or all 64 bits of a `long`, etc.

As a short example of how we use the logical operators, imagine that we have a circumstance on which we wish to model in our program. We are in control of a garage door. The door can be open or closed. At night, the door should be closed in order to help prevent burglars, but if it is raining, the door should be closed so that the rain doesn't get into the garage.

You could model the system above with a few variables: `boolean doorClosed;`, `boolean nightTime;`, and `boolean raining;`.

In order to determine whether or not the door should be closed, we might say something like `doorOpen = nightTime || raining;`.

This means the variable `doorClosed` should be set to the logical result of performing OR between the variables `nightTime` and raining. If it is `nightTime`, the `nightTime` variable will be 1, otherwise it will be false. If it is raining, the raining variable will be set to 1, otherwise it will be false.

We could add another condition. We could say that when it is nighttime, the door should be closed unless the light is on. Because if the light is on, perhaps that means that somebody is working the garage, which would look like this: `doorOpen = (!nightTime && !raining) || (nighttime && lightIsOn);`.

The preceding statement matches our conditions perfectly well, and the `doorOpen` variable will be set as we specified above. If the conversion from text to a Boolean expression looks daunting, keep in mind that there is no known easy way in mathematics or computer science to efficiently turn text into the simplest possible Boolean expression. Logical statements can be made arbitrarily complex, and it is deceptively difficult to accurately and efficiently describe the best possible method for representing some logical statements.

String concatenation

When we use + between two strings, it means to join them together. The term for this is concatenation.

Code Listing 6.10: String Concatenation

```java
String s1 = "This is";
String s2 = " a string";
String s3 = "!";
String concatenated = s1 + s2 + s3;
String str1 = "Concatenating integers is fine! " + 278;
String str2 = "And floats/doubles too!" + 45.678;
System.out.println(concatenated);
System.out.println(str1);
System.out.println(str2);
```

Code Listing 6.10 shows that concatenating integers, floating-point variables, and literals to strings is a matter of using the + operator. The `int` or floating-point literal will be converted to a string of characters before being added to the string. It is also possible to concatenate characters, such as A, or `boolean` literals such as false, to a string in the same way.

But, be careful: It is not possible to convert a string to an integer without calling the `Integer.parseInt` method!

Other operators

We will now examine two other operators that will make more sense when we look at control structures and inheritance hierarchies in the object-oriented section. Briefly, the conditional

operator is simply a shorthand for a particularly common if statement; and the instanceof operator is a mechanism for testing whether or not an object is an instance of a particular class.

Conditional operator

The conditional operator (sometimes called the ternary operator) is a shorthand way of writing a condition with two possible outcomes. In the chapter on control structures, we will see how to write if statements that perform exactly the same function as the functional operator. However, for now note that the conditional operator requires less code.

Code Listing 6.11: Conditional Operator

```java
public class MainClass {
    public static void main(String[] args) {
        int a = 25, b = 60;
        int larger = (a > b)? a : b;
        System.out.println(larger);
    }
}
```

Code Listing 6.11 shows the use of the conditional operator. The line `int larger = (a > b)? a : b;` is the invocation of the operator. The operator has three parts—a condition, a value when the condition is `true`, and a value when the condition is `false`. The operator will return either a or b, depending upon the result of the condition (a>b). In order to use the operator, we follow a conditional expression with a question mark, then place the outcome when the condition is `true`. After this, we use a colon, :, and place the outcome when the condition is `false`.

If we run the preceding program, we will see that the condition (a > b) is actually `false` because a is 25 and b is 60, and 25 is not greater than 60. Therefore, the value returned by the operator will be b, which we set to the larger variable and subsequently print to the screen.

The conditional operator above could easily be written out as an if statement, as in Code Listing 6.12 (we will look at if statements in the section on control structures).

Code Listing 6.12: Equivalent If Statement

```java
public class MainClass {
    public static void main(String[] args) {
        int a = 25, b = 60;

        int larger;

        if(a > b)
            larger = a;
        else
            larger = b;
```

```
            System.out.println(larger);
        }
}
```

instanceof operator

The final operator in Java is the `instanceof` operator. When we devise a class hierarchy, testing whether or not a particular object is of some class will often be useful. For instance, if we have a control on the screen and the user clicks on the control, we might want to know if the control is a button or not.

The operator takes two operands that are of some class hierarchy and determines if the left operand is an instance of the class specified by the right operand.

Code Listing 6.13: instanceof Example

```
public class MainClass {
    public static void main(String[] args) {
        MyClass someInstance = new MyClass();
        if(someInstance instanceof MyClass)
            System.out.println("someInstance is of MyClass!");
        else
            System.out.println("someInstance is not of MyClass...");
    }
}
```

Code Listing 6.13 shows a basic example of how to test whether or not an object, `someInstance`, is an instance of a particular class, `MyClass`. Note that this code will not run as a standalone application because I have not included the declaration of the class `MyClass`.

Tip: Avoid using the instanceof operator if possible. There are better ways to program structures that otherwise might use the instanceof operator. Let's say we need to program a method with this request: If the object is type t, do such-and-such; otherwise, if the object is of type q, do something else. In such a scenario, we might employ polymorphism in a more efficient manner. The compiler and JVM are able to detect and maintain object types, and we can specifically program our class hierarchies so that we never need to ask if object t is an instance of class c. We will examine class hierarchies in the section on object-oriented programming.

Challenges

Challenge 6.0: A farmer has collected 792,671 eggs from the chickens on his farm, and he wishes to sell them in cartons to commercial shopping outlets and supermarkets in regional north Queensland. How many full cartons worth of eggs does the farmer have (assuming 12

eggs fill a single carton)? How many eggs does he have left over for his own breakfast (hint: use the modulus operator)?

Challenge 6.1: Begin with the number four. Subtract two from it. Multiply this by 12. Increment your answer. Add your answer to itself. Divide it by seven and discard the remainder. Multiply your answer by itself. Cube your answer (i.e. raise it to the third power). Decrement your answer. What is the final result? The number you are left with is not divisible by one of the following primes—which one?

A) 2

B) 3

C) 17

D) 19

E) 43

Challenge solutions

Challenge 6.0

Code Listing 6.14: Solution to Challenge 6.0

```java
public class MainClass {
    public static void main(String[] args) {
        int eggsTotal = 792671;
        int cartons = eggsTotal / 12;
        int eggsLeftOver = eggsTotal % 12;
        System.out.println("The farmer has " + cartons + " cartons of
eggs, "
                    + "and " + eggsLeftOver + " eggs left over for
breakfast.");

    }
}
```

Challenge 6.1

Code Listing 6.15: Solution to Challenge 6.1

```java
public class MainClass {
    public static void main(String[] args) {
        int result = 4;
```

```
            result = result - 2;
            result = result * 12;
            result++;
            result = result + result;
            result = result / 7;
            result = result * result;
            result = result * result * result;
            result--;
            // To work out if a number evenly divides another, use mod:
            int mod2 = result % 2;
            int mod3 = result % 3;
            int mod17 = result % 17;
            int mod19 = result % 19;
            int mod43 = result % 43;
            System.out.println("The final result is " + result);
            System.out.println("Division by 2 leaves: " + mod2 + "
remainder.");
            System.out.println("Division by 3 leaves: " + mod3 + "
remainder.");
            System.out.println("Division by 17 leaves: " + mod17 + "
remainder.");
            System.out.println("Division by 19 leaves: " + mod19 + "
remainder.");
            System.out.println("Division by 43 leaves: " + mod43 + "
remainder.");
      }
}
```

Code Listing 6.15 produces the following output to the console when run:

```
The final result is 117648.

Division by 2 leaves: 0 remainder.

Division by 3 leaves: 0 remainder.

Division by 17 leaves: 8 remainder.

Division by 19 leaves: 0 remainder.

Division by 43 leaves: 0 remainder.
```

We can see from this output that division of our final result by 17 leaves a remainder of 8, but division by any of the other four primes leaves no remainder at all. This means that the final result is evenly divisible by 2, 3, 19, and 43, but it is not divisible by 17. In programming, there are usually many good ways to solve any particular problem. It is also possible to compute the result for Challenge 6.1 as a single expression (in fact, if we were programming this expression

in a real application, we would probably compute the entire thing first and only leave the literal answer). Code Listing 6.16 shows a few more examples of expressions that could solve this problem.

Code Listing 6.16: Other Solutions for Computing Challenge 6.1

```
// The first example shows computing the result without simplifying
// the expression:
int result1 = (((((((4-2)*12)+1)*2)/7)*(((((4-2)*12)+1)*2)/7))*
(((((4-2)*12)+1)*2)/7)*(((((4-2)*12)+1)*2)/7))*
(((((4-2)*12)+1)*2)/7)*(((((4-2)*12)+1)*2)/7)))-1;
System.out.println("Final result is " + result1);

// The second example shows that all we're really doing is raising
// 7 to the power of 6 and subtracting 1:
int result2 = 7*7*7*7*7*7-1;
System.out.println("Final result is " + result2);
```
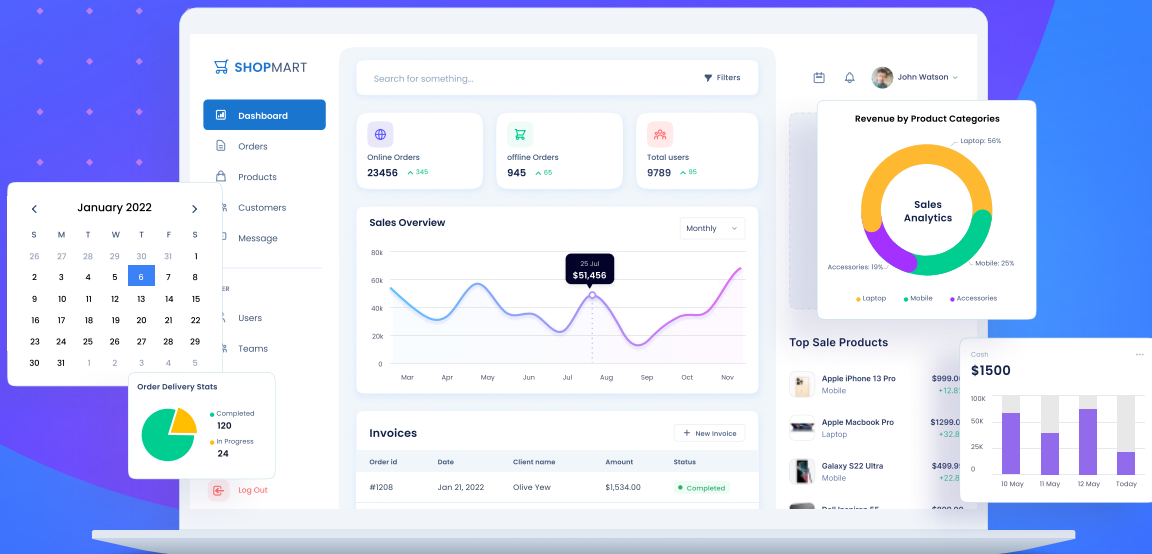
Chapter 7 Control Structures

Typically, computers execute code one line at a time from top to bottom. Control structures allow us to specify other orders of execution, such as branching on conditions and repeating loops of code. In Java, the available control structures are if statements, for loops, while loops, do while loops, switch statements, for each loops, and try/catch blocks.

If statements

If statements allow us to execute a block of code when a particular condition is true, otherwise the code block is skipped. The basic syntax to an if statement is listed in Code Listing 7.0.

Code Listing 7.0: Basic If Statements

```java
import java.util.Scanner;

public class MainClass {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int height = 0;
        System.out.print("How tall are you (in cm)? ");
        height = Integer.parseInt(scanner.nextLine());

        if(height < 100) {
            System.out.println("Wow! Are you a quokka?");
        }

        if(height > 300)
            System.out.println("Yeah right? Are you a giraffe?");

    }
}
```

Code Listing 7.0 shows two if statements. First, the listing shows a prompt to users asking for their height. The height variable's value is read as a string, using the scanner and parsed to an `int` with the `Integer.parseInt` method (parsing integers is not our primary focus, but this is an example of how it is done). The if statements begin with the `if` keyword and are followed by a `boolean` expression in brackets: (`height < 100`) in the first if statement, and (`height > 300`) in the second if statement (remember the relational operators < and > always return a `boolean` value). When the program evaluates the expressions, they will either be true statements or they will be false. When the statement evaluates to `true`, the code blocks following the if statement are executed. When the expression evaluates to `false`, the code block is skipped and the program will resume executing code after the if statement's code block.

If we run the code from Code Listing 7.0 and input a `height` of 20, the program will respond with `Wow! Are you a quokka?`. Running the program again, we could input a `height` of 500, and the program would output `Yeah right? Are you a giraffe?`. Note that if the input is greater than or equal to 100 and also less than or equal to 300, no output will be displayed.

When our code block consists of a single line of code (as in both of the examples from Code Listing 7.0), the curly brackets are optional. This is why the second if statement in Code Listing 7.0 is missing the curly brackets.

Combining several conditions together into logical expressions is often convenient. This is exactly what the Boolean operators ||, &&, and ! are used for. When we use || between two or more statements, any `true` statement will cause the entire expression to be `true`.

Code Listing 7.1: Using Boolean OR

```
int password = 25;
String userName = "Sam";
if(password == 10 || userName == "Sam") {
      System.out.println("That is the correct password or User Name!");
}
```

In Code Listing 7.1, the expression in the if statement is true if `password` is 10 or if the `userName` is `Sam`. This is Boolean OR, so the if statement is also true when both of these conditions are true (unlike English "or," in which "apples or oranges" means one or the other but not both). The logical OR operator combines the two statements `password==10`, and `userName=="Sam"` into a single `boolean` value. The operator returns true if either of the conditions is true or if they both are true. Typically, when we check a user name and password, we need both of them to be correct, so Boolean OR is probably not a good choice in this particular circumstance. Code Listing 7.2 shows the use of Boolean AND.

Code Listing 7.2: Using Boolean AND

```
int password = 25;
String userName = "Sam";
if(password == 10 && userName == "Sam") {
      System.out.println("That is the correct password AND User Name!");
}
```

The && operator combines two Boolean statements, and both must be `true` or the operator will return `false`. In the code, if the `password` is `10` and the `userName` is set to `Sam`, then the `System.out.println` will be executed. The password in the code is set to `25`, so the `println` will not execute. However, if you change the password to `10`, the `println` will execute because both conditions for the && operator are true.

If you want to check multiple possible conditions using the same variable, remember to use the variable name twice:

```
if(age == 50 || 60) { …                    // This is incorrect!

if(age == 50 || age == 60) { …   // This is correct!
```

The first example reads perfectly normally: If age is 50 or 60 …. However, this is not correct Java code. The computer will read the condition "age == 50". This expression evaluates to false. Due to the || operator, the program will look for the next condition. The next condition consists of the integer literal 60 by itself. But 60 is not a condition, and the program will not run. The correct way to check multiple settings for the age variable is illustrated in the second example—we include "age == 60" instead of 60 by itself.

Nesting if statements

We can place if statements inside the code blocks of other if statements. These are called nesting if statements.

Code Listing 7.3: Nested If Statements

```java
int age = 190, price = 14;
if(age > 50 && age < 100) {
      if(price > 20) {
            System.out.println("Age: " + age + " Price: " + price);
      }
}
```

Code Listing 7.3 contains a nested if statement. The first condition tests the variable "age" and determines if it is in the range of 50 to 100 (not inclusive). Due to the && operator, both conditions must be true. Therefore, if age is in the range from 51 to 99, the if statement's code block will execute. The inner if statement tests whether or not the price is greater than 20. This means that if the age does not fall within the range of 51 to 99, the computer will not consider the second if statement because it exists in the code block of the first.

Else if and else

We can follow an if statement with any number of else if (which translates into "otherwise") statements. And we can end a set of if/else if with a final else statement. Only one code block in a set of if/else if/else will execute. The code block that executes will always be the first block in which the condition evaluates to true.

If the condition of an if or an else if evaluates to false, the program will look for the next else if or else. When one of the conditions is found to be true, the computer will execute the associated code block for that particular condition, then it will drop down below the entire if/else if/else block and resume execution after the set of conditions. We use else if to provide additional conditions. The else if keyword is followed by a condition in brackets— exactly the same as an if statement. The difference is that an else if must have an if statement before it. The else if cannot stand alone.

The `else` keyword is always the final option. When the computer reads an `else` block, if none of the previous conditions evaluated to `true`, the `else` block will always execute. The `else` block is always placed at the end, and it does not have a condition.

Code Listing 7.4: If / Else If / Else Blocks

```java
int i = 25;
if(i % 2 == 0) {
      System.out.println("i is even");
}
else if(i % 3 == 0) {
      System.out.println("i is odd, but divisible by 3");
}
else if(i % 5 == 0) {
      System.out.println("i is divisible by 5");
}
else {
      System.out.println("i is not divisible by 2, 3, or 5");
}
```

Code Listing 7.4 shows a simple series of `if/ else if/else`. Only one of the conditions will have its code executed. The computer will check the `if` and `else if` from top to bottom. The variable `i` is set to 25, which is an odd number but is divisible by 5. Therefore, the second `else if` will print "`i is divisible by 5`". No other blocks will execute after this.

If, instead of 25, we set the variable `i` to 22, the first code block will execute and the program will print "`i is even`" to the screen. And, if we set the variable `i` to 17, note that 17 is not even, so the first condition fails. And 17 is not divisible by 3, so the second condition fails. And 17 is not divisible by 5, so the third condition fails. At this point the program encounters the `else`. The `else` block will always execute if the program encounters it, so with `i` set to 17, the program will print "`i is not divisible by 2, 3, or 5`" to the screen.

Loops

For loops

For loops allow us to execute a block of code a specified number of times. We often use a counter and execute the code until the counter reaches some value. This allows us to execute a code block a specific number of times. The syntax is as follows:

```java
for ([init]; [condition]; [increment] ) {
      // Body of the loop
      }

for([init]; [condition]; [increment])
      // Some single statement!
```

If the body of the for loop has a single statement, we do not need the curly braces.

Init: The initialization is where we declare the counter variable and set its initial value.

Condition: The condition is a Boolean expression, just like an if statement. The code block of the for loop will repeat until the condition evaluates to true.

Increment: The increment section allows us to increment or decrement our counter.

Code Listing 7.5: For Loop Counting from 0 to 9

```java
for(int j = 0; j < 10; j++) {
     System.out.println(j);
     }
```

Code Listing 7.5 shows a for loop that iterates 10 times and counts from 0 to 9. The initialization step of this loop declares and defines an integer variable called j, setting its initial value to 0. The condition is "j < 10", meaning "while j is less than 10." The increment is "j++", which means "every time the loop's body is executed, the j integer will count up." After executing the loop body 10 times, the j integer will increment to 10 and the condition "j < 10" will evaluate to false. The loop will terminate, and the program will continue to execute the code following the loop.

Note: Code Listing 7.5 shows a very common method for instructing the computer to loop 10 times. But notice that although the loop body is executed 10 times, the program counts from 0 to 9, not from 1 to 10. If you need your for loop to count as a human does, from 1 to 10, use something like the following instead: for(int j = 1; j <= 10; j++).

Code Listing 7.6: Counting Backwards with a For Loop

```java
for(int j = 10; j > 0; j--) {
     System.out.println(j);
     }
```

The for loop in Code Listing 7.6 counts backward beginning from 10 and counting to 1. It could be read "for j is 10, while j is greater than 0, decrement j". This is like a loop that counts up, but the "increment" part of the loop has "j--", which will cause the variable j to count downwards. Also, note that the starting value and the condition are switched—we start at 10 and loop while j is greater than 1.

Note: There is a special type of for loop used to iterate over the items in a collection, such as an array. The loop is commonly called a for each loop. For more information on the for each loop, visit
http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html.

Nested for loops

We can place a for loop in the body of another for loop. This is useful for traversing multidimensional arrays.

Code Listing 7.7: Nested For Loops

```java
for(int outerCounter = 0; outerCounter < 5; outerCounter++) {
    for(int innerCounter = 0; innerCounter < 3; innerCounter++) {
        System.out.println("Outer Counter: " + outerCounter +
                    " Inner Counter: " + innerCounter);
    }
}
```

Code Listing 7.7 shows nested for loops. The outer counter counts from 0 up to and including 4. In the body of the outer loop, the inner loop counts from 0 up to and including 2. The outer loop will execute five times, and each time it executes, the inner loop will execute three times. Therefore, the body of the inner loop will execute a total of 3*5 times, or 15. The output of this code is as follows:

Outer Counter: 0 Inner Counter: 0

Outer Counter: 0 Inner Counter: 1

Outer Counter: 0 Inner Counter: 2

Outer Counter: 1 Inner Counter: 0

...

Outer Counter: 3 Inner Counter: 2

Outer Counter: 4 Inner Counter: 0

Outer Counter: 4 Inner Counter: 1

Outer Counter: 4 Inner Counter: 2

Notice that the `outerCounter` remains the same for exactly three iterations of the inner loop. Then it counts up one. Each time the `outerCounter` counts up one, the `innerCounter` counts 0, then 1, then 2. You can nest for loops arbitrarily deep. We could have another for loop in the body of the inner loop in Code Listing 7.7 and another loop in the body of that one.

Break and continue keywords

The `break` keyword is used in the body of several control structures (`for`, `while`, `do while`, and `switch` as well). It causes the program to immediately jump to the next statement outside the body of the control structure. This is useful for stopping infinite loops (such as `while(true)`, which typically would never terminate) and for terminating the loop early.

The `continue` keyword is used to jump to the top of the loop. It is useful when you do not want the remainder of the loop's body to execute during some particular iteration.

Break and continue always operate on the innermost loop when loops are nested.

While and do while loops

There are two types of while loop—the while loop itself and the do while loop. The only difference is that the do while loop checks the condition after executing the body and is therefore guaranteed to run the loop's body at least once. Whereas the while loop checks the condition before running the body of the loop and, if the condition is false to begin with, the body of the loop will not execute at all. Code Listing 7.8 shows the basic syntax for these loops.

Code Listing 7.8: Nested For Loops

```java
while(condition) {
      // Body of the loop
}

do {
      // Body of the loop
} while(condition);
```

The condition can be any `boolean` statement (i.e. any logical expression that results in a `boolean` value). The loop will continue to execute while the condition evaluates to true. When the condition becomes false, the loop terminates.

Code Listing 7.9: While and Do While Loops

```java
int counter = 0;

while(counter < 10) {
      System.out.println("Counter in while: " + counter);
      counter++;
}

do {
      System.out.println("Counter in do ... while: " + counter);
      counter++;
} while(counter < 10);
```

Code Listing 7.9 shows an example of using a while and a do while loop. First, in the while loop, the `counter` variable will increment from 0 to 9. When the `counter` variable reaches 10, the condition of the while loop will fail. This particular while loop is very similar to a for loop, and in fact there is no while loop that could not also be written as a for loop and vice versa.

The do while loop has the same condition as the while loop. After the program executes the while loop, it will increment the `counter` variable to 10. The do while loop's condition is false

(`counter` is not less than 10). But the do while loop will still execute its body once before terminating the loop because it checks the condition at the end.

Switch statements

A switch statement is simply another way of coding a collection of if statements. In order to convert a collection of if statements into a switch, the conditions in the if and the else if statements must all address the same variable. In the following example code (Code Listing 7.10), all of the conditions address the variable `i`, therefore this series of if statements can be converted to a switch. The following shows a comparison of a collection of if/else and a switch statement that will do exactly the same thing (note that the switch example is not actually code and will not run).

Code Listing 7.10: If Blocks vs. Switch

```
if(i == 0) // Do something
else if(i == 1) // Do somethingElse
else if(i == 2) // Do somethingElse
else // Do the default

switch (i ) {
      case 0: { Do something; break; }
      case 1: { Do somethingElse; break; }
      case 2: { Do SomethingElse; break; }
      default: { Do the default; break; }
};
```

In a switch statement, we follow the `switch` keyword with a variable (`i` in the preceding example). Next, inside a code block, we provide a series of cases, each marked with the `case` keyword. When `i` matches a particular case, the code block for that case is executed.

In the switch example, each of the case's code blocks ends with the keyword `break`. This means that once a particular case has been executed, the program will break outside of the switch. If none of the previous cases is executed, the program will eventually find the case marked with the `default` keyword. The `default` case executes only if the previous cases did not. The `default` case is optional.

💡
Tip: The break keywords in a switch are not mandatory. If you do not place a `break` in the cases, the program will continue and find additional cases to execute. This functionality is seldom used, and I recommend that you always use `break` in every case of a switch statement in order to avoid confusion when other programmers (or you) read the code.

Code Listing 7.11 shows a practical example of a switch. The listing asks the user for two numbers and a mathematical operator. It uses a switch (highlighted in yellow) to decide what to do based on the operator the user selects (note that for simplicity, the program does not try to prevent dividing by zero).

Code Listing 7.11: Calculator with Switch

```java
import java.util.Scanner;

public class MainClass {
    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        // Parameters for the operation.
        double parameterA = 0.0, parameterB = 0.0;

        // Index of the operation.
        int operation = 0;

        System.out.println("Welcome to Calculator");
        do {

            // Ask the user for two numbers and an operator:
            System.out.print("Type a number: ");
            parameterA = Double.parseDouble(s.nextLine());
            System.out.print("Type another number: ");
            parameterB = Double.parseDouble(s.nextLine());
            System.out.println("Select an option: ");
            System.out.println(
"1. Add\n2. Subtract\n3. Multiply\n4. Divide\n0. Quit");
            System.out.print(": ");

            operation = Integer.parseInt(s.nextLine());

            // Perform the selected operation using switch.
            switch(operation){
            case 1: { // If the user selected 1, add the parameters.
                System.out.println(parameterA + "+" + parameterB
                        + "=" + (parameterA + parameterB));
                break;
            }
            case 2: { // If 2, subtract the parameters.
                System.out.println(parameterA + "-" + parameterB
                        + "=" + (parameterA - parameterB));
                break;
            }
            case 3: { // If 3, multiply the parameters.
                System.out.println(parameterA + "*" + parameterB
                        + "=" + (parameterA * parameterB));
                break;
            }
            case 4: { // If 4, divide the parameters.
                System.out.println(parameterA + "/" + parameterB
                        + "=" + (parameterA / parameterB));
                break;
```

```java
                }
                case 0: { // If 0, do nothing, they are quitting.
                    break;
                }
                default: { // Default option prints an error message.
                    System.out.println(
                                "That wasn't one of the options.");
                }
            }
        } while(operation != 0);

        // The user has quit.
        System.out.println("Bye, bye!");
        if(s != null)
            s.close();
    }
}
```

Try, catch, and finally

When something goes wrong in our programs, the JVM will throw an exception. This means it will halt execution of the application and possibly quit with an error message. This is not optimal behavior, and the keywords `try`, `catch`, and `finally` offer the programmer the opportunity to preempt possible problems while handling actual problems as they arise.

An `Exception` is a special type of object in Java. When a problem arises in our code, the JVM will create an `Exception` object, complete with an error message to describe what went wrong. The `Exception` object will be "thrown," which means the JVM will alert the current method that the exception has been encountered. The current method can either respond to the exception (as illustrated in Code Listing 7.12) or throw the exception again. If the current method throws the exception, the method that called it has the option of catching it and responding, and so on. If no methods respond to the exception, the program will crash and exit.

Code Listing 7.12: Try/Catch and Finally

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class MainClass {
    public static void main(String[] args) {
        float numerator = 0.0f;
        float denominator = 0.0f;
        Scanner s = new Scanner(System.in);

        // Try to execute some code block:
        try {
            System.out.print("Enter a numerator: ");
```

```java
            numerator = s.nextFloat();
            System.out.print("Enter a denomintor: ");
            denominator = s.nextFloat();
            if(denominator == 0.0f)
                    throw new IllegalArgumentException();
            System.out.println(numerator + " divided by " + denominator +
                            " is equal to " + (numerator / denominator));
        }

        // If an InputMismatchException is thrown
        catch (InputMismatchException e) {
            System.out.println("Your input was not a float.");
        }

        // If an IllegalArgumentException is thrown
        catch (IllegalArgumentException e) {
            System.out.println
            ("The result of division by Zero is not defined.");
        }

        // If any other type of exception is thrown
        catch (Exception e) {
            System.out.println
            ("An exception was thrown: " + e.getMessage());
        }

        // After we perform try and catch:
        finally {
            if(s != null)
                    s.close();
        }
        System.out.println("Thank you for your time.");
    }
}
```

Code Listing 7.12 asks the user for a numerator and a denominator for a division. If the user inputs valid floats for the numerator and denominator, the program will divide the numerator by the denominator and print the results. It will then print out a thank-you message and exit.

If the user does not input valid floats for the numerator or denominator (e.g., if the user inputs a string or inputs a denominator equal to 0.0f), the program prints an error message followed by a thank-you message and quits the application.

The program in Code Listing 7.12 shows how to use try, catch, and finally. First of all, try, catch, and finally each have a code block. Inside the body of the try block, we place the code we are attempting to execute. Notice that there is a line in this body of the try block that contains the throw keyword. When the denominator is 0.0f, this line of code will create a new IllegalArgumentException object, and this Exception object will be thrown.

Inside the `Scanner` class's `nextFloat` method, it checks that the value entered by the user is actually a `float`. If the user enters some string, e.g., `ninety four`, the `nextFloat` method will create and throw a new `InputMismatchException` object. Also note that the `InputMismatchException` requires an `import`—it is a class belonging to the `java.util` package.

Following a `try` block, we can include as many `catch` blocks as we like. Each `catch` block can be specifically designed to catch certain types of exceptions. The first `catch` blocks in a list of `catch` blocks should always catch the most specific exceptions because only one `catch` block will execute. In Code Listing 7.12, there are three `catch` blocks designed to catch `InputMismatchException`, then `IllegalArgumentException`, and then any other `Exception` object. The first `catch` block will catch the exception thrown by `s.nextFloat()` when the user does not input a float. The second `catch` block catches the `IllegalArgumentException` that we throw when the user attempts to divide by zero. The third `catch` block will catch any other exceptions thrown by the code that were not already caught by the previous `catch` blocks.

In the body of each `catch` block, we have the option of examining the exceptions that were thrown. In the final `catch` block, I have printed out the exceptions message by calling the `e.getMessage()` method. All exceptions have a `getMessage` method that can be used to show the user some extra information about what went wrong.

The `finally` block is optional. When used, it follows all the `catch` blocks. The `finally` block is always executed, regardless of whether or not an exception was thrown and caught—the code of the `finally` block will execute. I have used the `finally` block to close the `scanner`.

If our methods can potentially throw a particular exception and we do not offer any chance of catching and responding to the exception, we should mark the method with the throws keyword to indicate to any potential callers that this method may raise an exception. Code Listing 7.13 shows an alternative code that simply mentions that main throws exceptions while offering no `try/catch` to handle them.

Code Listing 7.13: Throws Keyword

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class MainClass {
    public static void main(String[] args)
                throws InputMismatchException, IllegalArgumentException {
        float numerator = 0.0f;
        float denominator = 0.0f;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a numerator: ");
        numerator = s.nextFloat();
        System.out.print("Enter a denomintor: ");
        denominator = s.nextFloat();
        if(denominator == 0.0f)
                throw new IllegalArgumentException();
        System.out.println(numerator + " divided by " + denominator +
```

```
                            " is equal to " + (numerator / denominator));

            System.out.println("Thank you for your time.");
        }
}
```

In Code Listing 7.13, the important line is highlighted in yellow. We state that our method potentially throws the two exceptions. This way, any method that calls this one might respond with their own `try/catch` blocks. As it happens, this is the main method, and the JVM itself called it to start our application. If this method throws an exception, the program will crash.

💡 *Tip: If you want to use a method provided by the Java libraries, such as `Scanner.nextFloat()`, you can hover over the `nextFloat` token in Eclipse and you will be shown a description of the method. Part of this description includes the exceptions that the methods could potentially throw and that you might want to include `catch` blocks for when you call the method.*

Study listing: Monty Hall program

Code Listing 7.14 shows an example of a simple guessing game. I have included this code so that folks new to programming can see how the control structures, variables, and other mechanisms of Java all go together to create a small game. This program introduces a few new concepts, such as the `Random` class (from math), but most of this program has been covered in detail.

Code Listing 7.14: Monty Hall Game

```java
// Import all classes from java.util.
import java.util.*;

public class MainClass {
    public static void main(String[] args) {

        // Declare a pseudorandom number generator called r.
        Random r = new Random();


        // Declare a new Scanner called scanner.
        Scanner scanner = new Scanner(System.in);

        // Select a random number from 1, 2, or 3.
        int doorWithCar = r.nextInt(3) + 1;
        System.out.println("There are three doors before you. " +
                    "Behind two doors, you will find a goat, but " +
                    "behind the third, there is a brand new car!\n");
```

```java
            // Declare a variable to store the door the user selects.
            int selectedDoor = 0;

            // Loop until they guess the right door:
            while (selectedDoor != doorWithCar) {
                System.out.println("Which door would you like to open?");

                // Read a door from the user.
                selectedDoor = scanner.nextInt();
                if (selectedDoor < 1 || selectedDoor > 3) {
                    System.out.println("The doors are 1, 2 or 3...");
                    selectedDoor = 0;
                } else if (selectedDoor == doorWithCar) {
                    System.out.println("Yes, you won a car!!!!");
                } else {
                    System.out.println("Nope, that's a goat...");
                    selectedDoor = 0;
                }
            }

        if(scanner != null)
            scanner.close();
        }
}
```

Line 1 shows how to import multiple classes with a single import statement. The star (or asterisk), "*", is used as a wild card that means import all of the classes from `java.util`. This is handy because in the code we use both a `Scanner` and a `Random` object. We could have used two import statements if we had preferred. The `Random` object is used to generate random numbers, and I have used it in Code Listing 7.14 to hide the car behind one of the three doors. `Random` does not actually generate a random number—computers cannot generate random numbers—but they can generate a sequence of numbers that will seem random (sometimes called pseudorandom numbers).

The following is an example of interaction with the Monty Hall program above.

```
There are three doors before you. Behind two doors, you will find a goat, but
behind the third, there is a brand new car!
Which door would you like to open?
19
The doors are 1, 2 or 3...
Which door would you like to open?
1
Nope, that's a goat...
Which door would you like to open?
2
Nope, that's a goat...
Which door would you like to open?
3
```

```
Yes, you won a car!!!!
```

This Monty Hall program is far more complicated than the previous programs we have studied, but it is a good example of how to create a coherent application from the basic building blocks of Java. Using only what we have studied so far, we can already make small games and other applications. The best possible way to learn a new computer language is to play around with the code, to read source code from other programmers, and to try figuring out why it does what it does. I have used some comments that will help you understand the objective of this source.

Challenge

Challenge 7.0: Fizz Buzz

Fizz Buzz is a common programming job interview question. It is easy to know what the different parts of a language do, such as a for loop, and an if statement. But this challenge is designed to test if a programmer is able to put the mechanisms together into a coherent working program.

The objective of this challenge is to print the numbers from 1 to 100 using print, except:

Every number that is evenly divisible by three, we should print "Fizz!" instead of the number.

Every number which is evenly divisible by five, we should print "Buzz!" instead of the number.

Numbers which are divisible by three and five, we should print "Fizz!Buzz!".

The output of your program should be as follows:

```
1 2 Fizz! 4 Buzz! Fizz! 7 8 Fizz! Buzz! 11 Fizz! 13 14 Fizz!Buzz! 16 17 Fizz! 19
Buzz! Fizz! 22 23 Fizz! Buzz! 26 Fizz! 28 29 Fizz!Buzz! 31 32 Fizz! 34 Buzz! Fizz! 37
38 Fizz! Buzz! 41 Fizz! 43 44 Fizz!Buzz! 46 47 Fizz! 49 Buzz! Fizz! 52 53 Fizz! Buzz!
56 Fizz! 58 59 Fizz!Buzz! 61 62 Fizz! 64 Buzz! Fizz! 67 68 Fizz! Buzz! 71 Fizz! 73 74
Fizz!Buzz! 76 77 Fizz! 79 Buzz! Fizz! 82 83 Fizz! Buzz! 86 Fizz! 88 89 Fizz!Buzz! 91
92 Fizz! 94 Buzz! Fizz! 97 98 Fizz! Buzz!
```

Tip: In order to discover if an integer j is divisible by another integer q, check if the remainder after division is 0 with the modulus operator. If the result of j%q is equal to 0, that means j is evenly divisible by q. For example, 20%4=0 because 20 is divisible by 4, but 81%6=3, and 3 is not 0, which means 81 is not divisible by 6. And, if an integer j is divisible by two integers, q and p, then j%(q*p)=0.

Challenge Solution

Challenge 7.0

There are many solutions to the classic Fizz Buzz problem. Code Listing 7.15 shows one.

Code Listing 7.15: Fizz Buzz Solution

```java
public class MainClass {
    public static void main(String[] args) {
        // Use a for loop to count from 1 to 100.
        for(int i = 1; i <= 100; i++) {
            // If the number is divisible by 3 and 5, print Fizz!Buzz!
            if(i % 15 == 0) System.out.print("Fizz!Buzz!");
            // Otherwise, if the number is divisible by 3, print Fizz!
            else if(i % 3 == 0) System.out.print("Fizz! ");
            // Otherwise, if the number is divisible by 5, print Buzz!
            else if(i % 5 == 0) System.out.print("Buzz!" );
            // Otherwise, just print the number itself.
            else System.out.print(i + " ");
        }
    }
}
```

Chapter 8 Object-Oriented Programming

Classes and objects

Java is an object-oriented language. It contains mechanisms that allow us to program hierarchies of our own data types called classes. A class is a blueprint—it is a description of a type of object in terms of which variables the objects own and which methods (functions) the objects are able to perform. We can create objects from these class blueprints—the objects are called instances of the class. Objects are a collection of variables called member variables and methods called member methods. For instance, you can create a `Circle` class with `radius` as a variable. And the methods might compute things such as `computePerimiter` and `computerArea`.

There are three key concepts for understanding the basics of classes and objects in Java:

- Member variables: describe what an object or class **has**.
- Member methods: describe what an object or class **does**.
- Inheritance: describes what an object **is**; what it inherits from parent classes.

Class keyword

In order to begin describing a new class, first add it to your project in exactly the same way we added the `MainClass` to our first project (click the File > New > Class). You must give your class a unique name in the New Java Class dialog box. We will first build a simple circle class, and I have named my class `Circle` in Code Listing 8.0. Note that the name of the class should match the name of the file, which means the file for this particular class is Circle.java (this will be automatic when we use the class wizard to add our classes to our projects).

Code Listing 8.0: Empty Circle Class

```java
public class Circle {
}
```

Local vs. member variables

We have defined many variables already, but in each case we have defined inside a method (specifically, inside the `main` method). Such variables have a scope (or visibility) that only lasts for a code block of the method. Variables that are defined inside a method and that have a scope that only lasts for the duration of the method are called local variables. In contrast to local variables, member variables have a scope that lasts for the lifetime of the object.

Member variables

Each of the objects built from a class has member variables. In order to create a member variable, we define the variable inside the class's body but outside of any methods. In addition to having a data type and name, member variables can also have an access modifier. Access modifiers describe the visibility for the member variable to the outside world, i.e. they define whether objects can access this member variable or whether it is private.

Code Listing 8.1: Circle Class Member Variables

```java
public class Circle {
      public float radius;
      private String name;
      protected float lineWidth;
      private static int circleCount;
}
```

In Code Listing 8.1, there are four member variables defined for our `Circle` class. Notice that there are terms next to the data types `public`, `private`, `protected`, and `static`. These are the access modifiers. They are placed to the left of the data type.

`public`: All classes have access to member variables and methods marked `public`.

`protected`: Classes and objects that belong to same package are allowed access to `protected` member variables.

`private`: Member variables marked `private` are only accessible by members of the class itself—that is, objects built from the class.

`static`: Members marked as `static` belong to the class rather than to the objects created from the class. If we mark a member variable as `static`, there will be one copy of the variable shared by all objects in the class. Members marked as `static` can also be referenced with the class instead of an object, as we shall see.

Note: The keyword $final$ is another access modifier. Members and local variables marked $final$ can have their value specified only once. The $final$ keyword is useful for defining constants such the mathematical constants PI and E. Final variables are usually named with all capital letters, such as PI and $DAYS\_IN\_JUNE$. For more information on the final keyword, visit https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html.

Instantiating an object

We have defined a class called `Circle`, and we can now begin constructing objects from it. Move back to the MainClass.java file in your project. Code Listing 8.2 shows how to create an object from our `Circle` class using the new operator.

Code Listing 8.2: Creating an Object from a Class

```
public class MainClass {
      public static void main(String[] args) {
            Circle c = new Circle();
            c.radius = 100.0f;
            System.out.println("Circle radius is " + c.radius);
      }
}
```

The most important line in Code Listing 8.2 is Circle c = new Circle();. This line creates a new Circle object from our class blueprint and assigns it to the object c. We begin with the class name, Circle, followed by the name we want for the object, c. Next, we use the assignment operator, =, followed by the new keyword. The new keyword is followed by what looks like a function call, but this is actually a very special function. Circle() is a constructor. It is a function used to initialize an object from a class. We did not supply any constructors in our Circle class, so Java has provided us with the default one, Circle() (we will define some constructors ourselves in a moment).

The object c is an instance of the Circle class. It owns a copy of each of the member variables specified in the Circle class. We can change the public member variables of c. In order to change the object's radius member variable, we use "c.radius = 100.0f". The period after the object's name means that the radius token belongs to the c instance. We can set the radius member variable as illustrated in Code Listing 8.2, but we cannot set the name or the circleCount member variables. If we try to, with "c.name = "Sirkl";" or "c.circleCount = 100;", you will notice that Eclipse underlines and states that these members are not visible because they are marked private and the main method is not part of the Circle class, therefore it cannot see these member variables.

Note: The main method is within the same package as the Circle class, so you will find that altering the lineWidth member variable is perfectly legal with "c.lineWidth = 23;". This is quite different from the protected modifier in C and C++ that means visible to child classes.

We can create many objects from a class. If we create a second Circle and call it b, the c Circle and the b Circle will both have their own member variables (see Code Listing 8.3).

Code Listing 8.3: Creating Multiple Instances

```
public class MainClass {
      public static void main(String[] args) {
            Circle c = new Circle();
            Circle b = new Circle();
            c.radius = 100.0f;
            b.radius = 900.0f;
            System.out.println("Radius of c: " + c.radius); // Prints 100
            System.out.println("Radius of b: " + b.radius); // Prints 900
      }
}
```

Code Listing 8.4 shows that we can very quickly create an entire array of object instances from our class. Note that when we declare an array of type `Circle`, all elements of the array are initialized to `null`. We must call the constructor for each instance in order to instantiate the objects.

Code Listing 8.4: Creating an Array of Objects

```java
public class MainClass {
    public static void main(String[] args) {
        // Declare array of 100 Circles objects.
        Circle[] circleArray = new Circle[100];

        // Use a loop to call new and set radiuses.
        for(int i = 0; i < 100; i++) {
            circleArray[i] = new Circle();
            circleArray[i].radius = (float)i;
        }
        // Print out a radius as an example.
        System.out.println
        ("circleArray[20] Radius: " + circleArray[20].radius);
    }
}
```

Member methods

Member methods are functions of a class. A function is a reusable block of code that we can invoke or call. In Code Listing 8.5, I have changed all the member variables to `private`. Marking all member variables as `private` and providing getters and setters as member methods (which enables us to alter the inner workings of the member variables with the knowledge that no external classes are accessing them directly) is common practice. The `Circle` class has also been expanded to include a number of member methods. The member methods included in Code Listing 8.5 are `setRadius`, `getRadius`, `print`, `zeroCount`, and `setLineWidth`.

Code Listing 8.5: Member Methods

```java
public class Circle {
    private float radius;
    private String name;
    private float lineWidth;
    private static int circleCount;

    // Constructors:

    // Default Constructor
    public Circle() {
        this.radius = 0.0f;
        this.name = "No name";
```

```
                this.lineWidth = 0.0f;
                circleCount++;
        }
        // Two argument constructor
        public Circle(String name, float radius) {
                this.radius = radius;
                this.name = name;
                this.lineWidth = 0.0f;
                circleCount++;
        }

        // Methods
        public void setRadius(float radius) {
                this.radius = radius;
        }
        public float getRadius() {
                return radius;
        }
        public void print(){
                System.out.println("Circle: " + name + " Rad: " + radius);
        }
        protected void setLineWidth(float newWidth) {
                this.lineWidth = newWidth;
        }
        public static void zeroCount() {
                circleCount = 0;
        }
        public static int getCount() {
                return circleCount;
        }
}
```

Member methods begin with access modifiers such as `public`, `private`, and `static`. These access modifiers behave in exactly the same way as when we mark member variables. The `public` designation means all external objects can invoke the method, while `private` means that only members of the class can invoke the method, etc. Following the access modifiers, we supply an identifier name for our method, then the parameters that the method requires.

The first two methods in Code Listing 8.5 are constructors. Constructors have no return type, and they share exactly the same name as the class. When we create an object from our class, we use the `new` keyword and call a constructor. The first constructor specified in Code Listing 8.5 takes no parameters. This is called a default constructor. The second constructor takes two parameters, a name and a radius, and sets the object's member variables based on the value of these parameters.

Note: Notice the use of the keyword `this` in the methods of a class. The keyword means "the current object." We can use the keyword `this` to differentiate between parameters being passed to a method and member variables with the same name. The

second constructor in Code Listing 8.5 shows an example of this—we pass a parameter called name and we use it to set the value of `this.name`, which is the member variable called name.

📝 *Note: When a member method or constructor takes more than one parameter, we separate the parameters in the parameter list with commas. This can be seen in the second constructor from Code Listing 8.5, which takes two parameters—a string called name and a float called radius.*

The methods `setRadius`, `getRadius`, and `setLineWidth` are called getters and setters. They are member methods that allow external objects to change the values of the corresponding `private` variables. A getter is used to read a variable's value. Getters take no arguments, and they return whichever type the variable happens to be. Getters generally consist of nothing but a single line that returns the variable's value. A setter is used to set the value of a private member variable. Setters typically take a single parameter of the same name and type as the variable they are used to set. Setters usually consist of a single line that is used to assign the new value to the member variable.

Calling member methods

After we supply a class with member methods, we can create an object and call the methods using the instance. Code Listing 8.6 shows how to create a `Circle` object and call the member methods specified.

Code Listing 8.6: Calling Member Methods

```java
public class MainClass {
      public static void main(String[] args) {
            // Create a circle with the default constructor.
            Circle c = new Circle();

            // Create a circle with the (String, radius) constructor.
            Circle b = new Circle("Dennis", 55.0f);

            // See how many circles we've counted with getCount.
            System.out.println("Number of circles created: " +
 Circle.getCount());
            b.print();         // Call the print method of b.

            c.setRadius(27.0f);// Call setRadius to set c.radius to 27.0f.
            c.setLineWidth(100.6f); // set c.lineWidth to 100.6f.
            System.out.println("Radius of C: " + c.getRadius());

            Circle.zeroCount();      // Calling a static method.
            System.out.println(
"Number of circles created: " + Circle.getCount());

      }
}
```

To call a member method, we supply the object's name, followed by the dot operator, followed by the name of the method (e.g., `c.setRadius(100.0f)`). In the brackets to a method call, we supply the parameters.

📝 *Note: In Java classes, we can have multiple methods with exactly the same names so long as each method takes a different number of arguments, or arguments with different data types, as input. This is called method overloading, and Code Listing 8.6 shows an example of overloading with the constructors. Notice that there are two methods called `Circle`, and they take different parameters. When we call one of these constructors, Java will know which one we mean by the parameters we pass. We can also overload nonconstructor methods—e.g., we have multiple `setRadius` methods: one that takes a `double`, one that takes an `int`, etc.*

Inheritance

Inheritance saves us time. We want to type as little code as possible, and we do not want to rewrite the same code more than once. Inheritance allows us to program classes with a parent/child relationship. The objective is to describe the attributes of a parent class in broadly applicable terms. From the parent class, we can create more specific child classes.

The following class is called `Animal`. It will be used as a parent class, and it describes aspects applicable to all animals. Animals possess voluntary motion, and they ingest some form of sustenance for energy, so I have included two `boolean` variables in Code Listing 8.7 and set them to `true` in a default constructor. You can add this new class to the existing project in the same way that we added the `MainClass` and the `Circle` class previously, or you might begin a new project.

Code Listing 8.7: Animal Parent Class

```java
public class Animal {
     public boolean voluntaryMotion;
     public boolean requiresFood;

     public Animal(){
          voluntaryMotion = true;
          requiresFood = true;
     }
}
```

An insect is a type of Animal. All insects have six legs, antennas, and an exoskeleton. Some insects have wings, so we will include a `boolean` for wings, and set it to true by default in the constructor. But because all insects are animals, we can inherit from the `Animal` parent class instead of adding the member variables for `voluntaryMotion` and `requiresFood`. Code Listing 8.8 shows the syntax for inheriting from the `Animal` parent class using the `extends` keyword.

```java
public class Insect extends Animal {

        public boolean antennas;
        public String skeleton;
        public int numberOfLegs;
        public boolean wings;

        public Insect(){
                super();
                antennas = true;
                skeleton = "Exosketon";
                numberOfLegs = 6;
                wings = true;
        }

        public void Print() {
                System.out.println("Antennas: " + antennas +
                                " Skeleton: " + skeleton +
                                " Number of Legs: " + numberOfLegs +
                                " Wings: " + wings +
                                " Voluntary Motion: " + voluntaryMotion +
                                " Requires Food: " + requiresFood);
        }
}
```

The most important syntax in Code Listing 8.8 is highlighted in yellow. The `Insect` class a derived class, or a child class, and `Animal` class is a super class, or a parent class. The first line of Code Listing 8.8 shows the use of the keyword `extends`. This keyword means the `Insect` class extends, or inherits, from the original traits specified in the `Animal` class.

Also notice the call to a method `super()` in Code Listing 8.8. This is a call to the `Animal` class's constructor. From a child class, we can use the term `super` to mean the parent class, and by supplying an empty parameter list, we are calling the default constructor of the `Animal` class. In Code Listing 8.8, we see that the default constructor sets the `boolean` variables `voluntaryMotion` and `requiresFood` to true.

Note: It is not necessary to call a parent's constructor in a child class, but if the parent's constructor is called, it must be called on the first line of the child class constructor.

Code Listing 8.9: Insect Class

```java
public class MainClass {
        public static void main(String[] args) {
                // Create an instance of Insect called i.
                Insect i = new Insect();
```

```
            // Change the insect class's members:
            i.antennas = false;
            i.numberOfLegs = 100;
            // Call the Insect's Print method:
            i.Print();
            // Change the parent class's requiresFood member:
            i.requiresFood = false;
        }
}
```

Code Listing 8.9 shows an example of creating an `Insect` object and calling some of its members.

We could then go on to define an `Ant` class. Ants are insects, therefore they have six legs, an exoskeleton, and antennas, but they do not have wings (at least not typically). Therefore, we can create a new child class called `Ant` that inherits from the `Insect` parent but that has the `wings` boolean set to false, as in Code Listing 8.10.

Code Listing 8.10: Ant Child Class

```
public class Ant extends Insect {
        public Ant() {
                super();
                wings = false;
        }
}
```

In Code Listing 8.10, notice that we call the parent constructor with `super()` before setting `wings` to `false`. The parent constructor must be called as the first statement of a child constructor. However, the parent constructor sets the `wings boolean` to `true`, and we want our ants to be wingless.

We could easily add more classes, such as `Mammals` and `Elephants`, to this hierarchy. Each time we add a new derived class, we need only specify how it differs from the parent or super class (Figure 42 is an example hierarchy).
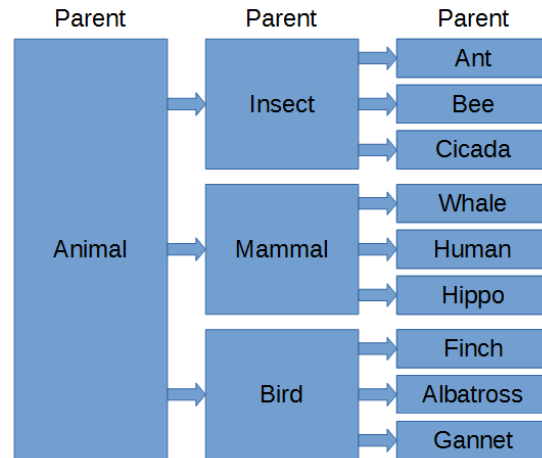
Figure 42: Animal Class Hierarchy

There is a lot more to object-oriented programming—such as abstract classes, interfaces, and polymorphism. These are advanced topics, and we will cover them in the next book, *Java Succinctly Part 2*.

Chapter 9 Example Programs and Conclusion

Our final chapter provides two case studies—Nim and the 3x+1 program. Each consists of complete, useful, and interesting programs that we can create with only minimal understanding of Java. The main difficulty in learning a computer language is gaining the skills to put together the mechanisms of the language into a coherent program. Finding complete, useful programs that only use the smallest subset of the mechanisms of a language is not easy. But the two examples in this chapter show that we do not need anything more than what we have seen so far—we can already create very useful programs.

In fact, while the game of Nim is a bit of fun, the 3x+1 program illustrates that we are already capable of exploring some of the most difficult unsolved problems known in mathematics and computer science. We might know only a tiny portion of Java, but the power that this language gives us is already extraordinary.

I have purposely left out the object-oriented mechanisms we looked at in the previous chapter. Object-oriented programming comes into its own when we look at polymorphism, which we will cover in *Java Succinctly Part 2*.

The game of Nim

The game presented here is a version of a simple and popular strategy game called Nim (for more on variants of Nim, visit https://en.wikipedia.org/wiki/Nim). You'll find a collection of sticks (the number begins at 37 in the program), and players take turns (beginning with the human player) to take one, two, or three sticks from the collection, reducing the collection of sticks for the next player. The objective of the game is to take the final stick.

A fun question might be—is there a strategy for beating the computer every time?

(Hint: Yes, there is! Upon careful inspection of the way the computer selects its moves, you might be able to discover how to win against the computer every time.)

Code Listing 9.0: Nim Source Code

```java
import java.util.Scanner;

public class MainClass {
    public static void main(String[] args) {

        // Declare local variables.
        Scanner scanner = new Scanner(System.in); // For reading input.
        String playerInput = ""; // Variable for reading player input.
        int sticks = 37; // Current number of sticks left.
```

```java
            // Print some instructions:
            System.out.println("Remove 1, 2 or 3 sticks from a pile.");
            System.out.println("The player who takes the final stick");
            System.out.println("is the winner!");

            // Main game loop:
            while(true) {
            // Tell the user how many sticks are left, and print a prompt.
                System.out.println("There are " + sticks + " left. ");
                System.out.print("How many would you like to take? ");

                // Read the user's input.
                playerInput = scanner.nextLine();

                // Subtract the number of sticks the player chose:
                if(playerInput.equals("1"))
                    sticks--;
                else if(playerInput.equals("2"))
                    sticks-=2;
                else if(playerInput.equals("3"))
                    sticks-=3;

            // If they did not select 1, 2 or 3, ask them to select again:
                else {
                    System.out.println
                    ("You can only take 1, 2, or 3 sticks...\n");
                    continue;
                }
                // Did the player win?
                if(sticks == 0) {
                    System.out.println("You took the last stick, you
win!");

                    break;
                }
                // If the player did not win, it's the computer's turn:
                if(sticks % 4 != 0) {
                    // Print the player's and the computer's moves.
                    System.out.println
                    ("Ok, you take " + playerInput + ", I'll take " +
                              (sticks % 4) + ".");

                    // Subtract the computer's move from the pile:
                    sticks -= (sticks % 4);
                }
                else {
                    // We have lost unless the player makes a mistake!
                    // Select a random number of sticks to subtract:

                    int take = (int)(Math.random() * 3.0) + 1;
```

```
                    // Print out the player's and computer's moves.
                    System.out.println
                    ("Ok, you take " + playerInput +
                            ", I'll take " + take + ".");

                    // Subtract the computer's move from the pile:
                    sticks-=take;
            }

        // If the computer just took the last stick, the player loses!
            if(sticks == 0) {
                    System.out.println("I took the last stick, I win!");
                    break;
            }
        }
    }
}
```

The program uses many of the ideas we've seen before, including `Math.random()`. Recall that `Math.random()` produces a pseudorandom number in the range from 0.0 up to but not including 1.0. The number generated is a double, so it could be something like 0.266787623 or 0.898898276. If we multiply the double by 3.0, we will get a random number in the range from 0.0 up to (but not including) 3.0. And recall that when we cast a double to an integer, the digits right of the decimal are discarded. Thus, when we convert our double to `int`, we will get a random integer 0, 1, or 2. And when we add 1 to this value, our integer will be 1, 2, or 3. Thus, when it is the computer's turn and the computer determines that it has already lost, it will select a random number of sticks in an attempt to throw off the player.

3x+1 program

Java is a big language, and we have only scratched the surface, but the reality is this—even beginner computer programmers have a rare power at their fingertips. In this section, we will look at a small program that might be employed by a mathematician to write a journal paper. We will explore one of the many unsolved problems in mathematics and examine how easy it is to explore such fascinating worlds of thought with minimal Java syntax.

The 3x+1 problem is a famous unsolved mathematical conjecture. It is a question relating to a simple game involving integers. The game is as follows:

Starting with a positive integer x, we must perform the following:

If x is even, divide it by 2.

If x is odd, multiply it by 3 and add 1.

Repeat the above steps until x reaches the number 1.

For example, if x begins as 7:

> 7 is odd, multiply by 3 and increment to get x = 22.
>
> 22 is even, divide it by 2 to get x = 11.
>
> 11 is odd, multiply by 3 and increment to get x = 34.
>
> 34 is even, divide it by 2 to get x = 17.
>
> 17 is odd, multiply by 3 and increment to get x = 52.
>
> 52 is even, divide it by 2 to get x = 26.
>
> 26 is even, divide it by 2 to get x = 13.
>
> 13 is odd, multiply by 3 and increment to get x = 40.
>
> 40 is even, divide it by 2 to get x = 20.
>
> 20 is even, divide it by 2 to get x = 10.
>
> 10 is even, divide it by 2 to get x = 5.
>
> 5 is odd, multiply by 3 and increment to get x = 16.
>
> 16 is even, divide it by 2 to get x = 8.
>
> 8 is even, divide it by 2 to get x = 4.
>
> 4 is even, divide it by 2 to get x = 2.
>
> 2 is even, divide it by 2 to get x = 1.

At this point, the value of x is 1, and the series for x = 7 terminates. We could continue the sequence even after we reach 1, but the value of x begins to loop through the sequence 1, 4, 2, 1, 4, 2… and never stops.

This game is very simple, but it brings up a question that has thus far thwarted all attempts at answering: Are there any starting values for x that do not lead to 1? That is: Could we set x to some positive integer so that it forms some other loop that does not involve 1, or so that the values of x do not loop but increase toward infinity with no boundary?

At present, the answer to the preceding question is not known. Mathematicians state this question as a conjecture (i.e. an unproven statement): There are no starting values for x that do not lead to 1. For more information, visit the Wikipedia page on the Collatz conjecture: https://en.wikipedia.org/wiki/Collatz_conjecture).

The following program asks the user for an integer. It proceeds to play the 3x+1 game until it reaches the number 1. At that point, it tells the user how many iterations were required to get to

the number 1 from the starting point, and it also prints out the largest value that x reached before receding to 1.

Code Listing 9.1: 3x+1 Program

```java
import java.util.Scanner;

public class MainClass {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String userInput = ""; // User's input string.
        int x; // The value of userInput converted to int.
        int iterations = 0;// Iterations to reach 1.
        int largest = 0; // Largest value reached in computation.
        while(true) {
            System.out.println("Enter a number, or 'Q' to quit: ");
            userInput = scanner.nextLine().toLowerCase();
            // Allow the user to quit:
            if(userInput.equals("q"))
                break;
            x = Integer.parseInt(userInput);

            largest = 0;
            for(iterations = 0; ; iterations++) {

                // If x is even, halve it:
                if(x % 2 == 0)
                    x /= 2;

                // Otherwise, x is odd, mul by 3 and increment.
                else
                    x = (3 * x) + 1;

                // Make sure we do not overflow the int:
                if(x <= 0) {
                    System.out.println(
                    "Overflow detected, use long for more range...");
                    break;
                }

                // If x is > than largest, record new record:
                if(largest < x)
                    largest = x;

                // If we find 1, we're done:
                if(x == 1)
                    break;
            }
            System.out.println("Reached 1 after " + iterations +
```

```
                " iterations. Largest point reached: " + largest + ".");
        }
    }
}
```

In Code Listing 9.1, I have selected to use `int` for the main data type of the variable `x`. We could extend the range of our program by using `long`, or even `BigInteger`, if we wanted. Because I have used `int`, the range that we can test is quite small (limited to about $2^{31}/3$), but it offers a good demonstration of how simple it is to create a genuinely useful and interesting application.

Challenges

Challenge 9.0 Is there any x value that does not reach 1?

Challenge 9.1 What is the pattern for the number of iterations as x increases?

Challenge 9.2 Is there some easy formula for predicting the number of iterations a given x value will take before it reaches 1, without actually playing through the 3x+1 game?

Challenge 9.3 Why does this simple game lead to such difficult questions?

The answers to these questions are unknown. Have fun exploring!

Conclusion

The information in this e-book covers the fundamentals of the Java language. We have looked at the primitive data types, variables, methods, creating classes, instantiating objects, and, briefly, at the beginnings of inheritance. We have covered control structures, keywords, and many other topics. But all of these topics only scratch the surface and hint toward the true power of Java, which is far more interesting and flexible than what we have seen so far. In the next e-book, we will cover many more practical aspects of the language, such as abstract classes, interfaces, polymorphism, creating graphical user interfaces, multithreading and much more.

I hope you have enjoyed this e-book, and I hope you'll look for the upcoming sequel, *Java Succinctly Part 2*.

Thank you for reading. Have a beautiful day!

Christopher Rose

Appendix: Keyword Reference

The following table lists most of the keywords reserved in Java at the time of writing. Future versions of Java occasionally implement new keywords. Several keywords are reserved but have no use, and I have not included these in the list.

Table 4: Java Keywords

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

Most of the keywords here are used very frequently, and the following is a brief summary of what each keyword does. Some keywords, such as `volatile`, `abstract`, and `interface`, will be explored in detail in the upcoming *Java Succinctly Part 2* e-book.

abstract: This is a modifier. It expresses that a class or method within a class is abstract. This means that the class cannot be instantiated as an object, but it can be used as a parent class. Any class with one or more methods marked with `abstract` must itself be marked as `abstract`.

assert: This is a debugging keyword used to test certain conditions. The keyword is followed by a `boolean` expression. If the Boolean expression evaluates to false, the program will terminate upon executing the `assert`. This allows us to test for particular conditions that will lead to errors in our code.

boolean: `boolean` is a primitive data type. This keyword is used to declare `boolean` variables and to cast to the `boolean` type. The `boolean` data type can be set to one of two values—`true` or `false`.

break: The `break` keyword is used to jump out of a loop or to switch/case block. If `break` is encountered inside a loop (for loop, while loop, do while loop), the program will immediately jump to the next executable line of code outside the loop's code block.

byte: This is a primitive data type. This keyword is used to declare `byte` variables and to cast to the `byte` type. The `byte` data type is used for small integers in the range from -128 to 127.

case: case is a selection mechanism. It is used with the switch keyword to create a list of possible cases in which the program might execute. Each case in a switch/case block is typically paired with a break, which means that program executes only a single case from a series of possibilities.

catch: catch is used in conjunction with the try keyword in order to specify a block of code that should be executed when a try block throws an exception.

char: char is a primitive data type. This keyword is used to declare char variables and to cast to the char type. The char type is used for characters—for example, A and &.

class: The class keyword signifies that the following code block is the declaration of a class.

continue: The continue keyword is used inside loops. When the program encounters continue, it will immediately jump to the top of the loop without executing any remaining code in the code block.

default: The default keyword is used in place of the final case in a switch/case block. If none of the previous case blocks execute, and a final default block is supplied, the default block will always execute.

do: The do keyword is used to signify a do while loop. The keyword is usually followed by a code block that forms the code to be executed in a loop. The code block of a do while loop ends with the term "while", followed by a condition. Do while loops are identical to while loops except that do while is guaranteed to execute at least once.

double: double is a primitive data type. This keyword is used to declare double variables and to cast to the double type. The double type is a 64-bit floating-point data type—it is used to represent approximations of numbers with a fractional part. The double has greater precision than a float, but it is twice the size in RAM.

else: The else keyword is used in conjunction with the if keyword to specify multiple possible paths of execution. The else keyword can appear by itself as the final option in a series of if and else if blocks. Or it can appear next to if in order to signify an extra condition.

enum: The enum keyword indicates that the following code block specifies an enumeration. Enumerations are a special type of class, specifically designed to represent sequential data such as the days of the week or the colors in the spectrum.

extends: The extends keyword is used in a class declaration to signify that the present class inherits from a parent class. See the section on object-oriented programming for more details.

false: The false keyword and the true keyword form the only two possible options for boolean variables and expressions.

final: The final keyword is used to signify that the following item will only be defined once. The final keyword is used to define constants—e.g., variables that we cannot change once they are set to some particular value.

finally: The `finally` keyword is used in conjunction with try/catch keywords to specify a code block that always executes. The `finally` code block is executed whether or not the try threw an exception (and thus began executing the catch block).

float: `float` is a primitive data type. This keyword is used to declare `float` variables and to cast to the `float` type. The float type is a 32-bit floating-point data type. It is used to represent approximations of numbers with a fractional part. The `float` data type has less precision than the `double`, but it is half the size in RAM.

for: The `for` keyword is used to specify a for loop. See the section on control structures for more information on for loops.

if: The `if` keyword is used to specify a condition under which the following code block should be executed. The `if` keyword, along with `else if` and an optional `else`, is used to offer multiple possible paths of execution that the program might take. See the section on control structures for more information on `if` statements.

implements: The `implements` keyword is used to specify that a class implements an interface. See the section on object-oriented programming for more information on the `implements` keyword.

import: The `import` keyword is used to import classes from an external package.

instanceof: The `instanceof` keyword is used to test whether or not an object is an instance of some particular class. See the section on object-oriented programming for more details on the `instanceof` keyword.

int: `int` is a primitive data type. This keyword is used to declare `int` variables and to cast to the `int` type. The `int` keyword is a 32-bit whole number. See the section on variables and data types for more information on `int`.

interface: The `interface` keyword is used to signify that the following code block declares an interface.

long: `long` is a primitive data type. This keyword is used to declare `long` variables and to cast to the `long` type. The `long` keyword is a 64-bit whole number. See the section on variables and data types for more information on `long`.

native: The `native` keyword is used to connect a Java program to external, native code. Using `native`, we can execute code that is compiled in different languages, such as C or Assembly language. This keyword is rarely used.

new: The `new` keyword is used to create objects from classes. See the section on object-oriented programming for more information on the use of the `new` keyword.

null: The `null` keyword is used to set an object without actually creating one. It is technically not a keyword but rather a literal. There are times when we might or might not have constructed an object from some class. We can use `null` to test if an object has been created or not.

package: The `package` keyword is used to create name spaces. Packages are a grouping mechanism. They enable us to group classes together to form some coherent collection of tools.

private: The `private` keyword is an access modifier. It marks a member variable or method as being `private` to a class. This means that outside objects cannot access the member.

protected: The `protected` keyword is an access modifier. It marks a member variable or method as being protected. This means that only objects within the same package are allowed access to the member. This is quite different from other object-oriented languages in which protected means that child classes can access the members.

public: The `public` keyword is an access modifier. It marks a member variable or method as being `public`. This means that external objects can access the member.

return: The `return` keyword is used to return control back to a method. The `return` keyword can be followed by a parameter, in which case this parameter becomes the return value of the method. The `return` keyword can also be used by itself when the method does not return a value.

short: `short` is a primitive data type. This keyword is used to declare short variables and to cast to the `short` type. The `short` data type is a 16-bit whole number. See the section on variables and data types for more information on `short`.

static: The `static` keyword is used to declare member variables and methods as belonging to the class rather than the instances of the class. Members that are marked as `static` can be accessed via the class without instantiating an instance.

strictfp: The `strictfp` keyword is used to mark a method or class as using a strict version of floating-point arithmetic. This is a feature designed to allow programmers to be certain that their floating-point computations will be exactly equal regardless of the machine the application is running on. The use of the `strictfp` keyword is rare.

super: The `super` keyword is used to reference a child's parent class. The parent class is called `super`, and in the child class's code we can call the `super`'s constructor by using `super(…)`.

switch: The `switch` keyword is used to introduce a switch/case block. It is a control structure that allows a single option to be selected from a series of possibilities, which is similar to a collection of if/else if/else statements.

synchronized: The `synchronized` keyword is used to mark methods as being exclusive to a single thread at a time. Objects executing `synchronized` methods will always execute the method one at a time rather than all the objects trying to execute the code at once with multiple threads.

this: The `this` keyword is used inside a method in order to allow an object to refer to itself. The keyword is often used to differentiate between a method's parameters and an object's member variables that share the same name.

throw: The throw keyword causes an exception to be raised. The program will look for a relevant catch block and can potentially exit the application with an error message if the exception is not handled.

throws: The throws keyword is used to indicate that a method will potentially throw a particular exception. This allows the caller of the method to safely surround the method call with a try/catch block.

transient: The transient keyword is used when serializing objects. To serialize an object is to convert it to a stream of bytes. When we mark a member variable as transient, it does not become serialized.

true: The true keyword and the false keyword form the only two possible options for boolean variables and expressions.

try: The try keyword is used to surround code that could potentially throw an exception. We use try in conjunction with catch and finally to gracefully handle possible problems while our program executes.

void: The void keyword is used to mark a method as not returning any data type. All methods require a return type (otherwise, they are constructors), and if a method does not return any data at all, the return type must be marked void.

volatile: The volatile keyword is used to mark member variables as potentially being modified by multiple threads. The keyword ensures that all access to the variable with multiple threads is safely coordinated by blocking more than one thread at a time.

while: The while keyword is used to specify a while loop, and it is also used to specify a do while loop. These loops are blocks of code that repeat indefinitely until some specific condition is false.