Decoding AP Computer Science A

Moksh Jawa

Decoding AP Computer Science A

Moksh Jawa

This book is for sale at http://leanpub.com/decodingapcomputersciencea

This version was published on 2016-02-28

Contents

Chapter 3 - Introduction to Java

3.1 Chapter Overview:

Now that we have a basic understanding of the fundamental of computer science, we can finally dive into what AP Computer Science is all about: creating computer programs using Java. In my opinion, the best way to learn computer science is to immediately apply concepts that you learn and look at various programs and figure your way out through them. The objective of this chapter is for you to come out of it knowing the basics of Java and being able to create a simple Java program to display some text on the computer screen.

3.2 Hello World Program:

The Significance of the Hello World Program

The Hello World program is a classic program that is often used as an introduction to any programming language, including Java. The functionality of the Hello World program is very simple. All it does is print out the phrase "Hello World!" on the computer. First, we will look at the code that goes into telling the computer to display the phrase and, then, we will walk through it line by line. You can see the Hello World program and its output in Figure 3.1. This program demonstrates many basic fundamentals of Java and makes it excellent for beginners.

Figure 3.1 | HelloWorld.java

```
1  // HelloWorld.Java
2  /* Text-printing program.
3    This comment is shown as a traditional comment. */
4
5  public class HelloWorld
6  {
7    // main method begins execution of Java application
8    public static void main( String[] args )
9    {
10     System.out.println( "Hello World!" );
11   } // end main method
12 } // end HelloWorld Class
```

Output

```
> run HelloWorld
Hello World!
```

Getting the Hello World Program on Your Computer

First of all, open up Dr. Java on your computer. You should have it downloaded. If you do not, please refer back to Chapter 2. Create a New File (File > New) and save it as HelloWorld.java. Next, type up the code, or text, in Lines 1 - 12 exactly as you see it in Figure 2.1. Make sure that you do not ignore even a single character because they are all essential to the program. In addition, make sure you account for indents and spacing.

Once you have typed up the program, click the "Compile" button in the toolbar. You should see a message in the "Compiler Output" section towards the bottom of the screen that says "Compilation completed." If you see any error, or message other than that, you have made a mistake in copying the code for Hello World to your Dr. Java.

After successfully compiling the program without any errors, click the "Run" button in the toolbar. Once you click the "Run" button, notice that the bottom of the screen displays two lines: "> run HelloWorld" and "Hello World!" The first line indicates that the HelloWorld Java program is being run and the second line is the output of the program being run. Therefore, the program ran successfully and it printed out some text. Once you have the program typed up in Dr. Java and working, we can start looking at the code overall and line by line.

Patterns in the Code

There are couple of things that you may immediately notice once you have finished typing up the Hello World program. The first thing you may notice is that there is an array of colors. Each color has some connotation and as we look at more and more programs, you will start to understand what each color represents. Similarly, you may also note a lot of spacing such as indentations and blank lines. This spacing usually serves one of two different purposes. It either makes the code a little bit more spread out and readable so that it can be easily revised and interpreted by humans or it is mandatory so that the computer execute it. As we go through each line, you will realize where the spacing is needed and where it isn't.

Comments

Figure 3.2

```
1 // HelloWorld.Java
2 /* Text-printing program.
3   This comment is shown as a traditional comment. */
```

Line 1-3

The first three lines of the Hello World program start off with what looks like just plain English. These lines are known as comments. Comments are basically sidenotes that programmers place in their programs so that they can have a better idea of exactly what's going on. In this case, comments are used to display the name of the program and its functionality. The amazing thing about comments are that they do not affect the program at all whatsoever so you can change any of the text within them or remove them completely and the functionality of the program will not be affected. Although comments do not seem very useful, they come in handy once you start developing larger and more advanced Java programs where it's difficult to keep track of the functionality of all of the code. It's good to build a practice of using comments. There are two widely used comments in Java and both are on display in these three lines: end-of-line comments and traditional comments.

An end-of-line comment is a comment that begins with //. As you can see on Line 1, the comment begins with // so therefore it is a end-of-line comment. An end-of-line comment can only span across one line and it can begin anywhere on the line including the middle (see Line 11 or 12).

A traditional comment differs from an end-of-line comment because it can span multiple lines which makes it useful for a particularly long comment. It starts with /* and ends with */.

Both types of comments serve the same purpose and differ in length and declaration. It's also important to note that // or /* */ need to be used to indicate a comment is being used otherwise that computer will try to interpret the text and that will cause an error.

Blank Spaces and Lines

Line 4

This line is not particularly important to the program as it's just simply a blank line. Extra spaces and blank lines are use to make computer programs more easier to read and they are, just like comments, ignored by the computer and do not affect the functionality of the program.

Declaring a Class

Figure 3.3

```
5 public class HelloWorld
6 {
```

Line 5-6

Line 5 is where we finally dive into the actual program - this is where the computer starts interpreting whatever is typed. This line is considered to be a class declaration because you, the programmer, are informing the computer about a class. We went over classes briefly in Chapter 2 and it's important to remember that they are the basic building blocks of every Java program and each program has one class.

Let's examine Line 5 word by word. First, let's look at "class", the second word in the line. "Class" is a keyword that lets the computer know that we will be declaring a class. Recall that a class is the basic building block of a Java program. Next, "public", the first word, is also another keyword but it has a different purpose. It's a little bit advanced so we'll go over it a little bit later in the book. Don't worry if these concepts seem confusing. We're just starting to look at them and you'll start to grasp them before you know. The last word in the line is "HelloWorld." It simply represents the name of the class. You may notice that the file name is HelloWorld.java and the class name is HelloWorld. This is not a coincidence. The file name and class name must match for a Java program to function. There are also some guidelines that you should follow while naming a class. The name of a class is considered to be an identifier, meaning that it can consist of letters, numbers, underscores, and dollar signs and cannot have any spaces or begin with a digit. In addition, class names and word within them are usually capitalized (`HelloWorld`). Some valid class names are `Moksh_Is_Awesome$1000` or `$Jawa`. On the other hand, `1000$Moksh_Is_Awesome` (starts with a digit) or `Jawa12 $50` (contains a space) are some invalid identifiers.

Next, in Line 6, all we see is a simple left brace, {. This signifies the beginning of the body of the class. A left brace is always used after a class declaration and can represent the start of the body of practically anything. Similarly, a corresponding right brace, }, marks the end of the body. It's also important to note that everything inside of the body, Lines 7-11, is indented. This is an example of one of the most important spacing guidelines in Java: all the text within a body is indented.

Declaring a Method

Figure 3.4

```
7   // main method begins execution of Java application
8   public static void main( String[] args )
9   {
```

Line 7-9

Line 7 is a review of what we just learned. It is an end-of-line comment as indicated by the green text color and the two slashes that begin the line. Recall that this line has no impact on the functionality of the Hello World program and it's simply there to let us know what's happening in the program - the purpose of the main method in this case.

Line 8 is where the fun of the Java program begins. Just like we declared a class in Line 5, we are declaring a method in Line 8. Remember that a method is a component of a class that performs tasks and returns information. Line 8 is the basic starting point for any Java program. Every class usually contains multiple methods including only one main method. A main method is a requirement or else the Java program will not function. When a Java program is run, the main method is executed and it either performs the tasks itself or instructs other methods to return information.

Let's take a look at Line 8 word by word. The first word you see is `public`, which, if you recall from the class declaration, is a keyword that we will be addressing later in the book. The same goes for `static`. The `void` keyword signifies that the method does not return information and we will be understanding its importance to methods a little bit later. Then comes the most important word in the entire line: `main`. Just like `HelloWorld` in the class declaration determined the name of the class, `main` does the same for the method name, which is also an identifier. Logically, the computer identifies the method with the name `main` as the main method. Lastly, `( String[] args )` is a required component of a main method declaration and we will go over why it's there when discuss methods more in depth.

Finally, on Line 9, we have yet another left brace and, as you probably guessed, represents the start of the body of the method. It was used in the exactly same way with the class declaration. Likewise, a right brace will follow later in the program to signify the end of the method body and the text inside body is indented.

Outputting Text in Java

Figure 3.5

```
9   {
10    System.out.println( "Hello World!" );
11  } // end main method
```

Line 10

Line 10 is the most important line of this entire program; it makes the HelloWorld program print out "Hello World!" Notice that it is indented as it is contained within the body of the method. This line contains many traces of concepts we have briefly touched but are yet to completely understand so don't worry if you feel a bit lost.

The first part of this line is `System.out`. `System.out` represents an object known as the standard output object. This object allows any Java program to display text. The System.out object has a

method called `println()` that prints a line of text. The "Hello World!" in between the parentheses represents what the method will print out. In this case, it is "Hello World!" but it can be replaced by anything (as long as it has quotation marks around it) and that will be printed out. For example, if Line 10 was changed to `System.out.println("Moksh is the best!");`, then the program would print out "Moksh is the best!" The phrase contained inside of the parentheses is known as an argument of a method and, soon, you will learn more about arguments and how important they are to Java programming. Lastly, there's a semicolon (;) after the closing parenthesis. It represents the end of a statement. Although it does not seem very significant, removing the semicolon will prevent the program from compiling due to an error. A semicolon, like indentation or braces, is another requirement of programming in Java.

Right Braces to End Classes and Methods

Figure 3.6

```
11    } // end main method
12  } // end HelloWorld Class
```

Line 11-12

For the last two lines of your first Java program, we finally see the much-awaited right brace, }. Remember that right braces complement the left braces by ending the bodies of a class or method. In addition, after the right brace closes the body, the following code no longer has to be indented like the text within the body. Following the right braces you can see two end-of-line comments which, just to reiterate, do not impact the program and simply let us know when the bodies of the HelloWorld class and main method are ending.

3.3 Importing Packages:

One of the primary advantages of Java is that it promotes software reusability thanks to the Java Class Libraries. There are thousands of classes available for programmers to use and, in this section, we will be exploring how these classes are organized and how programmers like you can use them in your Java program.

Packages

A package is a group of related classes. Remember that a class is like a blueprint and is the fundamental building block of Java programs. We also went over Java Class Libraries, or Java APIs, which are existing classes that any programmer can use while developing a computer program. The Java Class Libraries are filled with an enormous amount of classes and this is where packages come into play. Packages are used to organize, or group, the various classes based on their similarities. An

example of a package is the `java.lang` package which contains a lot of classes that are commonly used by Java programmers and essential to the majority of Java programs. Similarly, there are many other packages that Java programmers would like to use in their programs. However, an essential question comes up: how does a Java programmer use an external package in his or her Java program?

Import Statement

The answer to that question is to import the package using the import statement. Quite simply, the import statements allows you to import, or utilize, a package or a class that you require for your Java program.

Figure 3.7

```
1 import packageName.*;
2 import packageName.className;
3 import packageName.subpackageName.*;
```

We can see a couple examples of import statements in Figure 3.7. Using the import statement on Line 1, we can easily import an entire package. It's that simple. `Import` is the keyword that enables the importing to happen. `packageName` would logically be replaced with whatever the name of the package is (for example: `java.lang`). The asterisk, *, at following the package name tells the computer to import of all of the classes within the package. Lastly, there is a semicolon (;) at the end because this is a statement and Java requires that you put a semicolon at the end of statements.

There are also situations where you don't want to import all of the classes within a package and just need to use one. In that case. the import statement on Line 2 comes in very handy. The import statement is very similar to the one on Line 1 with the only difference coming from replacing the asterisk that enabled all classes to be imported with a name of a class within the package so only that class is imported.

Finally, there are some packages that are so big that they have to be split into subpackages first. An import statement for such a situation can be seen on Line 3. The only change is that the subpackage name follows the package name. After that, you can place an asterisk to import all of the classes in the subpackage or you can import a specific class in the subpackage.

You may also be wondering where the import statement fits into a Java program. Import statements are typically placed before the class declaration at the beginning of the program. Later in the chapter, we will be creating a Java program that takes in input from the user so we will use the import statement to import a class that will enable us to get user input.

3.4 Basics of a Java Program:

As we start to look at more and more Java programs, it would be helpful to know some of the basic requirements of a Java program so that it will be easier to understand. You may not understand some of these requirements until we dive deeper into Java but it's important to keep them in mind.

Firstly, as mentioned earlier, a Java program must have at least one class.The class usually contains a main method but remember that a class is required and a main method is not. Tracking back to the `HelloWorld` program, recall that it had a class called `HelloWorld`. A Java program does not exist with a class. Another concept to recall is the idea that Java is a high-level language and that a computer only understands machine language. To solve that problem, a compiler (built into Dr. Java) converts Java so that it can be interpreted by the computer. Finally, take a look at the code in Figure 3.8.

Figure 3.8

```
1  public static void main( String[] args )
2  {
3    // code
4    // code
5  }
```

The code seen above should be familiar to you. You saw it first in the `HelloWorld` program. Just in case you don't remember, this is a main method. Starting with Line 1, as we went over earlier, here a main method is being declaring. While looking at the declaration word by word, I mentioned that there were many words or phrases like `String[] args` would be explained later. Until we go over them, it's essential that you memorize this main method declaration because we will be using this declaration in almost every Java program we create in this book. Similarly, remember the left and right braces that open and close the body and that the body of the main method is indented.

3.5 Variables:

To understand variables, you have to look outside of computer science. In mathematics, a variable is a symbol that represents any particular value. For example, you know that `x + 2 = 3`. Based on that statement, you can determine that `x = 1`. `x` is a variable because it represents the value 1.

Likewise, in computer science, a variable also stores some arbitrary value. It also has a type, meaning that it can represent values like a number or some text. We will go over the various types for a variable in the next section. Another characteristic of a variable is that the value it stores can vary If `x` has a value of 1, its value can change if 1 is added to `x`. Its new value would be 2. Variables are an integral part of Java and computer science in general. They are used in most computer programs so make sure you understand their purpose and characteristics.

3.6 Primitive Types and Strings:

As we discussed in the preceding section, variables can store many different types of values and that every variable has a specific type. Well, those various possible types are known as primitive types, or built-in types. There are a total of 8 primitive types in Java: **int**, **double**, **boolean**, char,

byte, short, long, and float. We will be focusing on the 3 types that are in bold font: int, double, and boolean because they are the most commonly used types and are the types present in the AP Computer Science Java subset and exam.

The `int` Primitive Type

As you may have guessed, the `int` primitive type is short for integer. It's just like any integer that you may be accustomed to seeing in mathematics; it's a whole number that can be positive or negative. Some examples of an integer include 2, -600, 543, -2. However, in computer science, there is a slight restriction on the value of an integer. It can still be positive or negative but it must fall within a range of -2,147,483,648 to 2,147,483,647. Values beyond that range are considered to be `long`. It also makes sense that these other types for too large or too small values are not used much as the existing range encompasses most values that would come up in a Java program.

Finally, remember that variables that are of type `int` store numbers that are integers. Such variables are often used in programs to store values that represent an amount, total, or anything numerical. Because numbers make up such a big part of our daily lives and, hence, Java programs, `int` variables are fairly common.

The `double` Primitive Type

Once you understand the `int` primitive type, understanding the `double` primitive type is a walk in the park. Specifically, a `double` represents a double precision floating-point number. Although it sounds quite complicated, it's actually very simple; it's basically the same as an integer except it can be used as a decimal. Some examples of a `double` are 2.23, -3.567, -123.23, 345.545 and common uses for `double` variables are representing price, weight, height, and other numbers that are usually represented with a decimal and are not whole numbers.

The `boolean` Primitive Type

This primitive type is a little bit different from the types we have looked at so far. An `int` or `double` variable can essentially have an infinite amount of different values. On the other hand, a `boolean` variable can only have one of two values: `true` or `false`. This makes it one of the simplest primitive types to work with. You may be wondering where `boolean` variables fit into a Java program. There are often situations in Java programs where a variable represents the answer to a question like "Is the car working?" The answer to this question (`true` or `false`) can be represented using a `boolean` variable called isCarWorking and the value of the variable can be changed depending on what response to the question is. This primitive type will soon become extremely useful as we look at control structures and conditions later in this book.

Strings

Now that we have learned a bit about the major primitive types, let's talk about strings, which many people mistakenly assume to be a primitive type as well. A string is considered to be an object and we will discuss that in more depth down the road, but it's important to remember that a string is not a primitive type.

A String is just a set of characters, or text. You can identify a string in Java because it is enclosed by quotation marks (“). For example, in the `HelloWorld` program, we printed out a string (“HelloWorld!”). The double quotes around it tell the computer that it is a string. Strings are often used in our various Java programs because there is almost always text used when interacting with a user.

3.7 Declaring and Casting Variables

You now know the values that variables can store. The next step is to learn to tell the computer that we have a variable. In this section, we will learn how to declare a variable of any type and then will move onto to slightly more advanced topics of casting and final variables. You will see variable declarations come up in almost every Java program so make sure you understand how they are implemented.

Variable Declarations

Variable declarations are very similar to the class and method declarations that we have already looked at. They include the name of the variable and a keyword. In Figure 3.9, you can see two different variable declarations in Line 2 and 3.

Figure 3.9

```
1 // Variable Declarations
2 type variableName;
3 type variableName = value;
```

These declarations are relatively self-explanatory. For the declaration in Line 2, the `type` would be replaced by whatever primitive type, like `int`, that the variable represents. The `variableName` would logically contain whatever the name of the variable you would like it to be. A variable name is also an identifier so all of the rules for an identifier apply. Since a variable declaration is a statement, a semicolon (;) is used to conclude the line.

Line 3 is very similar to Line 2 because it contains everything in Line 2 along with `= value`. The purpose of the equal sign (=) is to assign a value to the variable. Following the equal sign, the `value` is whatever you'd like to variable to store as long as it is in correspondence with the primitive type that you have indicated in the beginning of the line.

You may have noticed that the variable declared in Line 2 did not have a value associated with it, unlike Like 3. In a regular Java program, you can later assign a value to the variable from Line 2. We will look into how we do that in the next subsection. As a recap, we use variable declarations to initialize, or create, variables and this forms the foundation of our Java programs.

Declaring Different Types of Variables

Figure 3.10

```
1  // Variable Declarations
2  int x = 2;
3  int y;
4  y = 5;
5  double pi = 3.14;
6  pi = 3.1415;
7  boolean isMokshCool;
8  isMokshCool = true;
9  String s = "This is a string!";
10 String s2 = new String("This is a string!");
```

Let's dive right into the code. Most of the code in Figure 3.10 should seem familiar to you based on the syntax we looked at in Figure 3.9.

Line 2 shows an `int` variable declaration. We know this because the line starts of with the word `int` in the spot where the primitive type of the variable should be. The name of the variable is `x` and it has a value of 2. That's how simple a variable declaration is - that's it!

Likewise, in Line 3 and 4, a variable called `y` is being assigned a value of 5 in a slightly different way. The variable is initialized in Line 3 but is not given a value. Then, in Line 4, the variable is set to a value of 5. Notice that in Line 4, the type is not indicated because it has already been declared when the variable was initialized.

Line 5 demonstrates another variable declaration of, in this case, a `double` variable. Since the variable is of type `double`, it can have a value of 3.14 (or any other decimal). If an int variable was assigned a value of 3.14, it would cause an error because integers can only be whole numbers. The variable declared in Line 5, pi, is then given a different value of 3.1415. The original value of 3.14 is replaced and longer represents "pi." In Java, you can easily change the value of variables by simply assigning a different value to them.

Lines 7 and 8 show a `boolean` variable being initialized and then set to a value of `true`. The only difference between these two lines and Lines 3 and 4 is that the primitive type of the variable being created is `boolean` rather than `int`.

Last but certainly not least, Lines 9 and 10 are also variable declarations, but they are different from other declarations. They are declarations for variables that are Strings. This difference is on account

of a String not being a primitive type. Line 9 follows the typical structure for the declaration with `String` in the place of a primitive type but notice that the primitive types like `int` or `double` are not capitalized while `String` is. Otherwise, the value of the variable is "This is a string!" which is enclosed by quotation marks just like any other String. Despite accomplishing the same job as Line 9, Line 10 deviates a bit more from what you would expect. In Line 10, everything on the left side of the equal sign remains the same buts it the right side where things start to change. As you can see, there is additional new keyword and the String value of the variable is enclosed in parentheses. Just like the other variable declarations we have looked at, there is a specific syntax that you have to follow. As mentioned earlier, both Line 9 and 10 fulfill the same purpose and you would logically prefer to use the variable declaration in Line 9 because it's shorter and more similar to a traditional variable declaration. The reason the declaration in Line 10 specifically applies to Strings has to do with a String being an object rather than a primitive type, a concept we will go over in more detail later down the road

Casting Variables

Casting variables refers to the concept of changing the primitive type of a variable or assigning a value to a variable of a primitive type that doesn't correspond. Two such examples of casting variables are when you want to assign an `int` value to a `double` variable and vice versa. One occurs smoothly while the other requires precaution while coding and some extra text.

Figure 3.11

```
1  // Casting Variables
2  int x = 2;
3  double y = x; // Fine
4
5  double x = 3.87;
6  int y = x; // Error
7  int y = (int) x; // Fine
```

Let's start off with the easier one: assigning an `int` value to a `double` variable. You can see an example of this in Lines 2 and 3. In Line 2, a variable called "x" is initialized and assigned a value of 2. The same happens on Line 3 with a variable "y" getting the value of "x", or 2. The variable "y" is a `double` but it's being assigned a value from a `int` variable. Remember that even though a `double` variable is used usually for numbers with decimals, there is no restriction on storing a value that is just a whole number. Therefore, assigning an `int` value to variable `y` does not cause an error at all.

Alternatively, storing a `double` value to an `int` variable causes an error as seen in Lines 5 and 6. Unlike a `double`, an `int` is meant only for whole numbers and giving an `int` variable `y` a `double` variable of 3.87 causes an error. This is where casting variables comes into play. On Line 7, you can see a solution that utilizes a cast. By adding in `(int)` between the equal sign and `x`, the computer converts the value of the variable `x` to an `int` value so that it can be stored to the variable `y`. You

would assume that the `double` value 3.87 would be rounded up to an `int` value of 4. However, the `double` value is truncated meaning that everyone to the right of the decimal is disregarded so 3.87 gets converted to an `int` value of 3 and is stored to the variable `y`. The `(int)` that we added is known as a cast and it enabled us to convert from a `double` value to an `int` one.

Figure 3.12

```
1  // Round Instead of Truncate
2  double x = 3.87;
3  int y = (int) (x + 0.5);
```

There are often situations in programs that require you to round, rather than truncate, the `double` value. In that case, add on 0.5 to the `double` value. The truncation will still happen but the truncated value of the `double` value plus 0.5 will be equivalent to rounding the `double` value. As you can see in the example above, the `double` value 3.87 has 0.5 added onto it so that it is 4.37 before it is truncated. After truncation, the new value is 4, making it correctly rounded. When working with negative `double` values, subtract 0.5 rather than add.

In conclusion, forgetting to cast is an error that throws off many Java programmers. It is essential so that we can merge values of different primitive types together. In this section, we looked at converting between `double` and `int` values but casting is not just limited to that. Casting can be used to turn numbers (`int`) into text (String) and so on. There are, however, some limitations such as the fact that you cannot convert a `boolean` value to an `int` value.

Final Variables

For the last part of this section, we will be looking at a different type of variable called a final variable. A final variable is also known as a constant. This means that once a value has been assigned to it, it cannot be changed. The variables we have encountered thus far can have their value modified later in the program but a final variable is different in that way.

Figure 3.13

```
1  final type variableName = value;
2  final int numOfPeople = 10;
```

The code above shows how we indicate that a variable is a final variable. We do so by putting the `final` keyword before we declare the primitive type of the variable; that's the only change that needs to be made. If we try to change the value of the `int` variable declared in Line 2 later in the program, there will be an error.

You are likely wondering why final variables are needed. Let's look at an example. Let's say that you are creating a Java program that is dependent on the number of people present and that number

is 10. There may be many locations in the code where the 10 has to be referenced as that is the number of people. Instead of putting 10 everywhere, it would be much better to put a variable, like `numOfPeople`. This also makes it much easier if the number of people present change because then you just have to change the value of the variable rather than every instance of 10 in the Java program. Similarly, there are countless different scenarios where constants are needed to represent a value and final variables are the solution.

3.8 Input and Output

Getting User Input in Java Programs

Input is something that is used in practically all computer programs. Input refers to the concept of the user entering, or inputting, information. Most computer programs have some element of interaction between the computer and the user so that the user stays engaged. Although input is not part of the AP Computer Science Java Subset, meaning that it will not show up in the AP Computer Science exam, it is essential part of Java and will come in handy as you start to create your own Java programs.

Figure 3.14

```
1  // Using Input
2  import java.util.Scanner; // import Scanner class from Java Class Libraries
3
4  // in main method
5  Scanner input = new Scanner( System.in ); // Scanner object
6
7  System.out.println("Enter the first number: ");
8  int num1 = input.nextInt(); // get 1st number
```

There are a couple of different things that you have to do in order to enable input in your Java program as seen in Figure 3.14. Input is also perfect example for us to use the Java Class Libraries that we discussed in Chapter 2 and the import statement from earlier in the chapter. You can see the import statement used in Line 2. In order to enable input, we use a class called the Scanner class that is part of the `java.util` package. Remember that it's an import statement so you will see it before the class declaration in the first few lines of the Java program.

After importing the `Scanner` class, you need to create an object of the `Scanner` class once you are in the body of the main method. You can accomplish this using the code in Line 5 which looks fairly similar to a String variable declaration because both the String is an object just like the Scanner object. Essentially, we are declaring a variable called input that refers to a `Scanner` object, or an instance of the imported `Scanner` class. A variable like such is known as an object reference. This

object reference declaration is something that you just have to remember because you will use it in your programs that involve input.

Now that we have instantiated an object of the `Scanner` class, we can call the methods within the `Scanner` class. The primary method within the class that we will be utilizing is the `nextInt()` method. The method name does a good job of explaining its purpose, or functionality. It essentially reads whatever `int` value the user inputs next. In Line 7, the user is prompted to enter in a number and then the `nextInt()` method is called in the following line. In addition, in Line 8, a variable is initialized and it stores whatever the `nextInt()` method returns, essentially what the user inputs. Pay close attention to how the method is called or run. The format for is `object.methodName()`. Whenever you call methods, this is the syntax to follow. In addition, there are many methods within this class, such as `nextDouble()` or `nextLine()`, that function similarly to `nextInt()`. We will use input for a Java program later in this chapter.

Outputting Text in Java Programs

Unlike input, output, rather than taking user input, display information for the user. Other differences include that it's included in the AP Computer Science Java subset and that we've already worked with it in the `HelloWorld` program. Just like input, output falls under the umbrella of keeping the user engaged and informed about what is happening in the computer program.

Figure 3.15

```
1  // Using Output
2  System.out.print("I love ");
3  System.out.print("AP Computer Science!");
4
5  System.out.println();
6  System.out.println("I love ");
7  System.out.println("AP Computer Science!");
```

Output

```
I love AP Computer Science!
I love
AP Computer Science!
```

Above, you can see an example of using some output methods to display text. You may notice that there is no import statement or object reference declaration in the code. Just as input relies on a Scanner object, output depends on a Standard Output Object. Fortunately, we don't have to import the Standard Output class or instantiate it because it is built-in due to its popularity.

There are two different methods in Standard Output class that are included in the Java subset and are most-used. These methods are `print()` and `println()`. You used the `println()` method in the `HelloWorld` program. Let's compare and contrast these two methods as they serve the same purpose with minimal differences. They are both methods of the `System.out` class. In addition, they both are used to print out, or display, text. As you saw in the `HelloWorld` program, a string or variable is put in between the parentheses to determine what will be printed out. The string or variable printed out is known as an argument that is passed to the method. The implementation of both methods is the same. There is only a slight difference between the two. You can see an example of the difference in the code above. If you look carefully, you will see that the `print()` method continues printing text on the same line while the `println()` method begins printing on a new line after it is executed. You can use either method depending on what your requirements for output are.

Escape Sequences

Printing out text also brings up another concept of escape sequences. An escape sequence is a combination of a backslash and a character and is used for formatting what is printed out by the computer program. When utilizing escape sequences, you tend to include them within the String that is being printed out and the computer automatically recognizes them. In the table below, you can see some widely-used escape sequences.

Common Escape Sequences | Figure 3.16

Escape Sequence	Purpose
\n	Continues printing on a new line
\\	Prints out a backslash
\"	Prints out a double quote
\t	Continues printing one tab to the right

Don't worry if this seems a bit overwhelming. Escape sequences are quite easy to use and we will see them being implemented in a bit. The `\n` and `\t` escape sequences are used for formatting what you print out and come in quite handy. The `\\` and `\"`, however, serve a slightly different purpose. Because the computer requires double quotes around Strings and does not print out the quotation marks, a programmer would be stuck in a tough situation if he or she actually needs to print out quotes. By inserting the \" escape sequence within the String and quotes, the computer will know that it needs to actually print out a quotation mark as well. Similarly, you could need to print out a backslash (\) but the computer may interpret it as part of an escape sequence (and, hence, not print it out). The `\\\` escape sequence solves that problem. There are a countless amount of escape sequences and those are some of the common ones.

Figure 3.17 | EscSeq.java

```java
// Escape Sequences
public class EscSeq
{
  public static void main( String[] args )
  {
    // \n escape sequence
    System.out.println("I\nlove this book");

    // \\ escape sequence
    System.out.println("\\ is a backslash");

    // \" escape sequence
    System.out.println("\" is a quote");

    // \t escape sequence
    System.out.println("I\tlove this book");
  }
}
```

Output

```
> run EscSeq
I
love this book
\ is a backslash
" is a quote
I        love this book
```

Figure 3.17 shows a Java program called `EscSeq.java` that demonstrates some escape sequences in action. As you compare the code and output, you can see the functionality of each escape sequence. In addition, the escape sequences are integrated within the quotes enclosing the String and are very simple to use. Overall, escape sequences are frequently used with output to format text and display special characters.

3.9 Arithmetic Operators

Arithmetic operators are something that we have been introduced to ever since we learned 1+1 at a very young age. They are used to represent arithmetic operations like addition. As you may know, computer science is based on a lot of mathematics and arithmetic operators are frequently used in Java programs to perform calculations.

Basic Arithmetic Operators

Most of the arithmetic operators that we will work with will be familiar to you and all of them should be very easy to use. Logically, arithmetic operators are used with numbers like an `int` or a `double`. In the table below, you can see the symbols for arithmetic operators, their corresponding operations, and examples of their use in both Java and algebra.

Arithmetic Operators | Figure 3.18

Symbol	Purpose	Algebraic Example	Java Example
+	Addition	x = y + 2	x = y + 2;
-	Subtraction	x = y - 2	x = y - 2;
*	Multiplication	x = 5y	x = 5 * y;
/	Division	x = $\frac{y}{5}$	x = y / 5;
%	Remainder	x = y mod 5	x = y % 5;

You should be familiar with the first four symbols, which just represent the basic arithmetic operations of addition, subtraction, multiplication, and division. As you may have noticed, arithmetic operations within Java are very similar to those in algebra. The primary difference is that an asterisk (*) represents multiplication and that a slash (/) does division. Speaking of division, division between integers works differently than you would imagine. For example, if you are doing `6 / 4`, you would assume the answer to be 1.5. However, Java requires that dividing two integers should result in a quotient that is also an integer. The correct answer, 1.5, is then truncated and the answer results to 1. The simple solution to this to use the `(double)` cast as follows: `(double) 6 / 4`. This will return an answer of 1.5.

The last operator, called a modulus, is a percent sign (%) and is not as common as the other operators. The use of the modulus is quite simple and is somewhat related to division. It essentially returns the the remainder of division between two numbers. For example, `10 % 3 = 1`. The modulus is particularly useful for determining divisibility because if a modulus returns a `0`, you know that they are divisible.

Increment and Decrement Operators

The increment and decrement operators a very simple purpose: to add or subtract 1 from a number. The increment operator (++) can make the equation `x = x + 1;` much simpler as `x++;`. Similarly, the decrement operator(– –) can do the same when subtracting numbers. In the table below, you can see how the use of increment and decrement operators varies a bit.

Increment and Decrement Operators | Figure 3.19

Symbol	Name	Example	Explanation
++	Prefix increment operator	++x;	Increase *x* by 1 and then use its new value in the same expression
++	Postfix increment operator	x++;	Use the current value of *x* and then increase it by 1
– –	Prefix decrement operator	– –x;	Decrease *x* by 1 and then use its new value in the same expression
– –	Postfix decrement operator	x– –;	Use the current value of *x* and then decrease it by 1

Putting the operator in front or behind the increment or decrement operator can make a world of difference. Let's consider an example. Say you have a variable `a` that is equal to 3 and you want to print it out a number that is 1 greater than it. You can utilize an increment operator for that, but your placement of the operator can determine whether you successfully complete your objective or not. If you decide to type `System.out.println(a++);`, then the computer will output 3 because it will use the current value of a first and then add on the 1. So if you were to try to print out a again after this statement, the computer would print out 4. Alternatively, if you used `System.out.println(++a);`, 4 will be outputted because a is increased first and then it is printed out. The same goes for decrement operators.

Once we start using control structures, we will use increment and decrement operators more than ever. In addition, these operators come in handy when keeping count because you tend to increase or decrease by 1.

Addition Program

In this subsection, you will look at a basic Java program that applies a lot of what we have learned recently. This includes input, output, and arithmetic operators. Below, you can see the code for the `Addition` program filled with comments along with the output of the program. Take a look at the code and try to interpret it to the best of your abilities. Then, read the explanation of the code that follows.

Figure 3.20 | Addition.java

```
1  // Addition.java
2
3  import java.util.Scanner; // import Scanner class from Java Class Libraries
4
5  public class Addition
6  {
7    public static void main(String[] args)
8    {
9      Scanner input = new Scanner( System.in ); // Scanner object
10
```

```
11      int num1; // first number
12      int num2; // second number
13      int sum; // sum of the first and second numbers
14
15      System.out.println("Enter the first number: ");
16      num1 = input.nextInt(); // get 1st number
17
18      System.out.println("Enter the second number: ");
19      num2 = input.nextInt(); // get 2nd number
20
21      sum = num1 + num2; // get sum by adding the 2 numbers
22
23      System.out.println("The sum is " + sum );
24    }
25  }
```

Output

```
> run Addition
Enter the first number:
2
Enter the second number:
3
The sum is 5
```

It definitely is a lot to take in. Don't be overwhelmed - we will work through this together part by part. By the time we finish, you will realize that creating a Java program and implementing the concepts you have learned is easier than you think. If we go over a concept and you don't remember much of it, track back to an earlier section in the back which will help you refresh your memory.

Line 1 - 3

Figure 3.21

```
1  // Addition.java
2
3  import java.util.Scanner; // import Scanner class from Java Class Libraries
```

The first 3 lines feature a few end-of-line comments that, as you know, don't affect the functionality of the program and simply keep us in the loop about what is going on. The most important part of this set of lines is the import statement in Line 3. We actually saw this line of code earlier in this very chapter when we went over input. Essentially, we are importing the `Scanner` class so that we

can instantiate a `Scanner` object. We will then be able to call methods from the `Scanner` class that will enable us to read whatever the user inputs. Remember that import statements come before the class declaration at the beginning of the program.

Line 5 - 8

Figure 3.22

```
5  public class Addition
6  {
7    public static void main(String[] args)
8    {
```

We essentially looked at this set of code with the `HelloWorld` program. In Line 5, we have a class declaration for the `Addition` class and then the signature right brace to signify the start of the body of the class. Then, on Line 7 and 8, the main method is declared along with the right brace and we are ready to start typing up code to make this `Addition` program come to life.

Line 9

Figure 3.23

```
9      Scanner input = new Scanner( System.in ); // Scanner object
```

In Line 9, we are creating a `Scanner` object and creating an object reference (basically a variable) called `input` to it. Remember that we are able to instantiate this `Scanner` object because we imported the `Scanner` class earlier in the program. Now, with the `Scanner` object, we can call methods that we need from the `Scanner` class.

Line 11 - 13

Figure 3.24

```
11     int num1; // first number
12     int num2; // second number
13     int sum; // sum of the first and second numbers
```

This is the first time you are coming across variable declarations in an actual Java program and, as you can tell, they are relatively straightforward. With these 3 lines of code, we are initializing 3 `int` variables named `num1`, `num2`, and `sum`. Notice that the variables are not given a value yet and are defaulted to a `null` value. Later in the program, when the time is right, we will assign a value to each one. The end-of-line comments following each variable declaration help explain the purpose of each of the variables. The variables `num1` and `num2` represent the first and second number that the user inputs and `sum` depicts the sum of the two numbers given by the user. Now that all variables are initialized, we will move on and assign the appropriate values to them.

Line 15 - 19

Figure 3.25

```
15      System.out.println("Enter the first number: ");
16      num1 = input.nextInt(); // get 1st number
17
18      System.out.println("Enter the second number: ");
19      num2 = input.nextInt(); // get 2nd number
```

This is where a lot of the magic happens. First, in Line 15, we prompt the user to enter the first number. Then, using the `nextInt()` method, we can get whatever `int` value that the user inputs when prompted. Then, we can set the value of the `num1` variable to whatever the user gives us. Similarly, we repeat the same process to obtain the second number from the user and then store it to the `num2` variable. The user has now done his or her job of giving us two numbers to work with and it's up to us to turn those two numbers into one.

Line 21 - 25

Figure 3.26

```
21      sum = num1 + num2; // get sum by adding the 2 numbers
22
23      System.out.println("The sum is " + sum );
24    }
25  }
```

If you are given two numbers and asked to find their sum, you add them. Similarly, we need to have the computer do the same. We will accomplish this using the arithmetic operators. The `num1` and `num2` variables are added together using the `+` arithmetic operator and their sum is assigned to the `sum` variable.

At this stage, all of our variables have been assigned a value and we have the final answer, or the sum, that we need to return to the user. The only task left is to display it for the user, which we will do using the `println()` method. In Line 23, we print out what the sum is. Rather than just printing out the variable, we place some text before the number so that the user understands the concept. The `+` operator is used again but for a different purpose: to combine the variable and String so that they can be outputted. In the output, you can see that `The sum is 5` is printed out.

Finally, there are two consecutive right braces to close the bodies of the main method and `Addition` class and that's it! We just walked through a Java program which interacts with the user and performs calculations. As you learn more concepts down the road, you will be able to add more functionality to create a better experience for the user.

3.10 Relational and Equality Operators

Although you may not know them by name, you always use relational operators when comparing different numbers. Examples of relational operators include the greater than sign (>) and the less than sign (<) and they are used in Java just as you are accustomed to using them.

Relational Operators

All of the relational operators that we will be working with will be entirely familiar to you and even more simpler to use in your Java programs. These operators are primarily used when you want to compare numbers like an `int` or a `double`. In the table below, you can see each relational operator along with its corresponding example Java condition and meaning.

Relational Operators | Figure 3.27

Operator	Name	Java Condition Example	Condition Meaning
>	Greater than	x > y	x is greater than y
<	Less than	x < y	x is less than y
>=	Greater than or equal to	x >= y	x is greater than or equal to y
<=	Less than or equal to	x <= y	x is less than or equal to y

The table does an adequate job of explaining the functionality of each operator and these are the very same operators that you use when working with inequalities in math. As you know, these operators are used based on the relationship between the two numbers and which is bigger or smaller. The only difference between the relational operators in Java and what you are used to is the way a few of them are represented. The greater than or equal to and less than or equal to operator are typically written as a greater than or less than sign with an underscore underneath them. However, in Java, they are written as >= or <=. Other than that, you can continue to use them in Java just as you have for the most of your life.

Equality Operators

Equality operators are actually very similar to relational operators and, as you can probably tell from the name, they are all about the equal sign. They are also used to compare numbers to each other but rather than checking if numbers are greater or less than each other, they check if numbers are equal to each other. Here is another table that explores the two equality operators that we will work with in this book.

Equality Operators | Figure 3.28

Operator	Name	Java Condition Example	Condition Meaning
==	Equal to	x == y	x is equal to y

Equality Operators | Figure 3.28

Operator	Name	Java Condition Example	Condition Meaning
!=	Not equal to	x != y	x is not equal to y

These 2 operators may seem a bit more unfamiliar to you. You may be used to seeing the not equal to operator (!=) as an equal sign with a slash through it but, because we do not have the ability to type characters with a slash through them on a computer, we use != as the alternative. Likewise, you are likely to the equal to operator consisting of only one equal sign rather than two. However, when you are trying to compare two different numbers you need to use two signs and when you are assigning values to variables use one. The == operator is an equality operator and the = operator is an assignment operator, which we will go over in the next section.

Use in Programs

You may be wondering where relational and equality operators fit into programs. These operators are used in a condition, or a boolean expression that can evaluate to `true` or `false` when comparing primitive types. In the two tables that we looked at earlier, the Java condition example column was filled with Java conditions that could evaluate to `true` or `false` depending on the values of `x` and `y`.

Figure 3.29

```
1  boolean x = ( 5 >= 4 ); // true
2  boolean y = ( 3 == 2 ); // false
```

In the figure above, the expressions enclosed within the 2 sets of parentheses are conditions because they evaluate to a boolean value. In the first line, the condition `5 >= 4` evaluates to `true` because 5 is indeed greater than 4. Hence, the `boolean` variable `x` is set to a value of `true`. Likewise, upon evaluation, the variable `y` is set to `false`. You can see that a relational operator and an equality operator are used in the condition. In addition, the single equal sign (=) is used to assign a value and the double equal sign (==) is used to compare values.

Comparison Program

Now we will apply what we have learned in this section but creating a `Comparer` program that prompts the user for 2 numbers and then tells the user if the first number is greater than the second number or not. You will use a relational operator to compare the numbers and print out the answer. Before you look at the code right below, think about how you would go about implementing such a program using what you have learned.

Figure 3.30 | Comparer.java

```java
// Comparison.java

import java.util.Scanner; // import Scanner class

public class Comparer
{
  public static void main( String[] args )
  {
    Scanner s = new Scanner( System.in); // Scanner object

    int num1; // 1st number
    int num2; // 2nd number

    System.out.println( "Enter the first number: " );
    num1 = s.nextInt(); // get 1st num

    System.out.println( "Enter the second number: " );
    num2 = s.nextInt(); // get 2nd num

    System.out.println("1st number greater than 2nd number: " + (num1 > num2));
  }
}
```

Output

```
> run Comparer
Enter the first number:
10
Enter the second number:
3
Is 1st number greater than the 2nd number: true
```

Line 1 - 18

The majority of this program is the same as what we saw in the `Addition` program so we are going to skip over it. Quickly glance over the first 18 lines and, if you come across anything that you don't understand, go back to the `Addition` program and check out the explanation.

Line 20

Figure 3.31

```
20      System.out.println("1st number greater than 2nd number: " + (num1 > num2));
```

Line 20 is really the only line in this program that contains something we haven't seen in a full-fledged program yet. At this point, we have 2 variables that store the `int` values that the user has inputted. Now, we need to tell the user if the first number is greater than the second number. We do this by using the greater than relational operator within a condition. The condition returns `true` or `false` which is then outputted by the computer to give the user the answer.

3.11 Logical and Assignment Operators

In the last section, we looked at relational and equality operators which are used when comparing numbers. Now, we will be working with logical and assignment operators which are another set of operators that are essentials to Java programs. We will be moving into control structures in the next chapter and you will find them to be especially useful.

Assignment Operators:

Assignment operators do exactly what they sound like: they assign values. We have already worked extensively with the most used assigned assignment operator - the equal sign. As you know, the equal sign is used so frequently because it is used whenever you are setting a value to a variable. The other assignment operators are similar to increment and decrement operators in the notion that you can live without them but they make life much easier. Check out this table below to learn more about the assignment operators.

Assignment Operators | Figure 3.32

Operator	Java Example	Explanation
=	x = 7	value of x is 7
+=	x += 5	x = x + 5
-=	x -= 3	x = x - 3
*=	x *= 9	x = x * 9
/=	x /= 2	x = x / 2
%=	x %= 4	x = x % 4

Because you already know the purpose of the equal sign (=) assignment operator, we will focus on the last 4 assignment operators. To understand their purpose, let's think of a situation where you have an `int` variable x and you want to add on 2 to it. You would have to add the expression `x = x + 2;` to your Java program. Instead, you can use the assignment operator += to simplify the expression

to `x += 2;` and it accomplishes the same objective. Similarly, you can place any of the arithmetic operators we know before the equal sign and it will work. These assignment operators will become very useful as you start to create more advanced Java programs.

Logical Operators:

Recall that in the last section we worked with conditions and the relational operators within them. Well, as a programmer, you can often come across situations that require you to evaluate multiple conditions at once and return a single `true` or `false` value. You can use logical operators to link these various conditions together. This may sound very abstract and confusing so let's just dive right into these operators and you'll see their use as we use them in situations.

The 3 logical operators we will work with in Java are `&&`, `||`, and `!`. First, let's look at the `&&` operator, which is also known as the logical AND operator. This operator is used to combine conditions together and it returns only if both conditions individually evaluate to true. Basically, the following complex condition `( 3 > 2 && 2 <= 5 )` would evaluate to `true` because both conditions 3 > 2 and 2 <= 5 are `true`. If even one of the conditions had been `false`, the complex condition would have come out to be `false`. In the table below, you can see that you can only get a `true` value from a complex condition with the *&&* operator if both conditions are `true`; otherwise, it evaluates to `false`.

&& Operator Truth Table | Figure 3.33

&&		
	T	**F**
T	T	F
F	F	F

With the `&&` operator, only 1 of the 4 possible situations result in a `true` value, but, with the `||` operator, 3 of 4 situations do. The `||` operator is also known as the logical OR operator and it is used just like the `&&` operator for complex conditions. The only difference lies in the notion that only 1 of the 2 conditions have to be `true` for the overall result to be `true`. This means that you can only get a `false` value from a complex condition with the `||` operator if both conditions evaluate to `false`. The following complex condition `( 3 < 2 || 4 > 1 )` would result in `true` because the second condition is `true`. Just as we saw a table for the possible outcomes with the `&&` operator, there is a table below that shows the same for the `||` operator.

|| Operator Truth Table | Figure 3.34

||		
	T	**F**
T	T	T
F	T	F

The last logical operator is a little bit different from the ones we have looked at thus far, but it also has many applications within computer science. This is the logical NOT operator that is represented by `!`, an exclamation mark. It simply reverses the value of a `boolean` expression. For example, the following condition `!( 3 > 2 )` would return `false` because 3 is greater than 2 and `!` operator results in the opposite of the value of the `boolean` expression. An exclamation point commonly represents opposite in computer science as we say with the does not equal operator (`!=`). You can see another table below for the outcomes with a logical NOT operator.

! Operator Truth Table | Figure 3.35

!	
T	T
F	F

These logical operators may not seem very relevant at this point, but, as soon as we hit the next chapter, they will become a commonplace in our programs. Keep them in mind and make sure you remember each one.

Short-Circuit Evaluation

Short-circuit evaluation is an issue that comes up as a product of using the logical AND and OR operators. To best understand short-circuit evaluation, let's pretend we have a complex condition that is `(false condition && true condition)`. As you know, both conditions have to be `true` in this situation to get a `true` value. When the computer evaluates this condition, it goes from left to right and, after it realizes the first condition is `false`, it skips over the second condition because it is impossible to end up with anything other than a `false` value. Likewise, the computer could be working with a complex condition like `( true condition | | false condition)` and the same thing would happen with the `| |` operator. Immediately after recognizing that the first condition was `true`, the computer would disregard the second condition because the overall result will definitely be `true`.

You may be wondering what's wrong with that. After all, all it does is help the computer evaluate a complex condition faster. However, it is possible that a condition that is skipped over by the computer actually impact the value of a variable. For example, let's say that the second condition of a complex condition is `x++ < 8`. Then, if it is disregarded by the computer due to short-circuit evaluation, `x` will never have 1 added on to it which can affect its overall value and create problems for later in the program. Yes, most conditions don't change values but short-circuit evaluation can create problems in certain conditions.

The solution to this is to use `&` and `|` instead of `&&` and `| |`. By simply, only put one symbol instead of two, the computer knows to evaluate both conditions no matter what the result of the first condition is. For example, if you have a complex condition like `(false condition & true condition)`, the computer knows that the overall condition will be false after evaluating the first one. But, because

there is a single & rather than two, it checks the second condition anyway. The same goes for using | instead of | |. Short-circuit evaluation often catches programmers off guard and will definitely show up on the AP Computer Science exam.

3.12 Order of Operations

We've looked at so many different operators in this chapter and parts of our programs can become especially confusing when there are many types of operators used. The compiler follows a specific order of operators to interpret; it doesn't just go left to right. It's important to know the order of operations because your calculations could return answers that you would not expect. In the table below, you can see the precedence of each operator we've learned of thus far.

Operator Precedence | Figure 3.36

Priority	Operator(s)	Associativity
1	++, – –, !	right to left
2	*, /, %	left to right
3	+, -	left to right
4	>, >=, <, <=	left to right
5	==, !=	left to right
6	&&	left to right
7	||	left to right
8	=, +=, -=, *=, /=, %=	right to left

Let's quickly glance through this table together and then, later in the chapter, you will have the opportunity to evaluate some expressions with various operators to check your knowledge of the order of operations. Associativity is referred to as the order in which the operator is evaluated. You can see that the most and least important operators are the ones that are from right to left while the rest are left to right. Otherwise, the table is rather straightforward and you just have to remember this order because it will become very important when you make complex calculations with your Java programs. The types of operators from most to least precedent is increment/decrement and logical NOT operators, multiplication/division/remainder, addition/subtraction, relational operators, equality operators, logical AND operator, logical OR operator, and assignment operators.

3.13 Chapter Summary

Wow! This is one long chapter. We went over so much and it definitely is a lot to taken. However, soon all of these concepts will become second nature to you just as concepts from Chapter 2 have. We looked at operators, input/output, declaring variables of different primitive types, and so much more. It's amazing that we have created so many different programs throughout this chapter and start off unable to develop a single one. Here's to more chapters just like this one where you learn a

lot!

Multiple Choice Questions

1) Which of the following class names is incorrect?

(A) `APcs_2015_java`

(B) `Apcs_java_2015`

(C) `Java_APcs_2015`

(D) `2015_java_APcs`

2) Which of the following comments is incorrect?

(A) `///AP computer science java.`

(B) `// AP computer science java.`

(C) `///AP computer science.`

(D) `/AP computer science/`

3) Which of the following comments is correct?

(A) `/*ABCD*/`

(B) `*/ABCD*/`

(C) `/*ABCD/*`

(D) `**ABCD//`

4) Which of the following `HelloWorld` programs is correct?

```
(A)       public class HelloWorld
    {
      public static void main(String[] args)
      {
        System.out.println(“Hello world”);
      )
    }
```

```
(B) public class HelloWorld
    {
      public static void main(String[] args)
      [
        System.out.println(“Hello world”);
      ]
    }
```

```
(C) public class HelloWorld
    {
      public static void main(String[] args)
      {
        System.out.println(“Hello world”);
      }
    }
```

```
(D) public class HelloWorld
    (
      public static void main(String[] args)
      {
        System.out.println(“Hello world”);
      }
    )
```

5) Do comments affect the functionality of the program?

(A) Yes, only while exectuing the class

(B) No, not at all

(C) Yes, only while executing the main method

(D) No, only the methods are affected

6) Which of the following main method declarations is correct?

(A) `public static void main(String() args);`

(B) `public static String main(String[] args);`

(C) `public static void main(String arg);`

(D) `public static void main(String[] args);`

7) Which of the following statement(s) is a valid variable declaration and assignment?

(A) `int x = 21.7;`

(B)
```
int x;
x = 21.7;
```

(C) `int x = (int) 21.7;`

(D) `int x = (double) 21.7;`

8) What is the result of the following expression?

```
int result = 2 - 6 / 3 * 4 + 2 % 5;
```

(A) `4`

(B) `-4`

(C) `3`

(D) `-3`

9) What is the result of the following expression?

```
int result = 5 / 3 + 6 - 2 * 2;
```

(A) 4

(B) -4

(C) 3

(D) -3

10) Consider the following code segment.

```
public class Test
{
  public static void main(String[] args)
  {
    int result1 = 2 - 6 / 3 * 4 + 2 % 5;
    int result2 = (2 - 6) / 3 * (2 + 2);
    System.out.println(result1);
    System.out.println(result2);
  }
}
```

What numbers are printed as a result of executing the code segment?

(A) Both numbers are the same

(B) -4, -1

(C) 4, 1

(D) -3, -1

11) Which of the following code segments properly declares and assigns the variable `x` a value?

I. `int x = 2;`

II. `int x = 20000;`

III. `int x = 21.2;`

IV. `double x = "X";`

V. `String x = "X";`

(A) Only I

(B) Only I and II

(C) I, II, III, V

(D) I. II, V

12) If `x` is a `double` variable with a value of 654321 and `y` is a `double` variable with a value of 654321.0, what will result from the following expression?

```
x == y
```

(A) `true`

(B) `false`

(C) 0

(D) Compile time error

13) Consider the following code segment.

```
boolean x = true;
boolean y = false;
boolean z = !x;

System.out.println("Result 1: " + (x | y));
System.out.println("Result 2: " + (y & z));
System.out.println("Result 3: " + (!z));
```

What is printed as a result of executing the code segment?

```
(A) Result 1: true
    Result 2: true
    Result 3: true


(B) Result 1: true
    Result 2: false
    Result 3: true


(C) Result 1: true
    Result 2: false
    Result 3: false


(D) Result 1: false
    Result 2: false
    Result 3: true
```

14) What occurs if an attempt is made to compile and execute the following code segment?

```
double big = 45.67;
int small = 45;
boolean result = (big > small && small != 100);
System.out.println("The result is " + result);
```

(A) `The result is true`

(B) `The result is false`

(C) An error will occur at Line 3

(D) An error will occur at Line 4

15) Which of the following variable declarations will not compile successfully?

(A) `String x = 5;`

(B) `double temperature = 15.24;`

(C) `boolean ok;`

(D) `String x = "5";`

16) Upon execution of the code fragment below, what will the values the variables `a`, `b`, and `c` be?

```
int a;
int b = 5;
int c = 3;
int a = --b * c++;
```

(A) a = 16, b = 4, c = 4

(B) a = 42, b = 5, c = 8

(C) a = 48, b = 5, c = 8

(D) a = 12, b = 4, c = 4

17) Consider the following code segment.

```
int g = 3;
System.out.print(++g * 8);
```

What is printed as a result of executing the code segment?

(A) `24`

(B) `12`

(C) `32`

(D) `16`

18) Consider the following code segment.

```
int g = 3;
System.out.print(g++ * 8);
```

What is printed as a result of executing the code segment?

(A) `16`

(B) `24`

(C) `8`

(D) `32`

19) Upon execution of the code fragment below, what will the values the variables `a`, `b`, and `c` be?

```
int a;
int b = 5;
int c = 3;
int a = b-- * c++;
```

(A) a = 16, b = 4, c = 4

(B) a = 42, b = 5, c = 8

(C) a = 35, b = 6, c = 7

(D) a = 15, b = 4, c = 4

20) Which of the following statement(s) is an invalid variable declaration and assignment?

(A)
```
int a = 10;
```

(B)
```
int a;
    a = 10;
```

(C)
```
a = 10;
```

(D)
```
int a;
```

Answers and Explanations

Answer Key

1. **D**
2. **D**
3. **A**
4. **C**
5. **B**
6. **D**
7. **C**
8. **B**
9. **C**
10. **A**
11. **D**
12. **A**
13. **B**
14. **A**
15. **A**
16. **D**
17. **C**
18. **B**
19. **D**
20. **C**

Explanations

1) (**D**) The name of a class is considered to be an identifier, meaning that it can consist of letters, numbers, underscores, and dollar signs and cannot have any spaces or begin with a digit. In addition, class names and word within them are usually capitalized (like `HelloWorld`). In this case, notice that all choices don't have any spaces and use the appropriate characters but Choice D starts with a digit, making it invalid.

2) (**D**) An end-of-line comment is a comment that begins with // and spans only one line. The contents of a comment do not matter since a comment does not affect a the functionality of a program. Since Choice D is does not begin with //, it is incorrect.

3) (**A**) A traditional comment is a comment that can span multiple lines and it is implemented by starting with /* and ending with a */. Choice A meets that requirement.

4) (**C**) All of the answer choices have the correct text and the only difference between them lies in how the bodies of the `HelloWorld` class and `main` method are enclosed. The bodies of a class and

method should be enclosed by braces, meaning that they start with `{` and end with `}`, which Choice C correctly does.

5) (**B**) Comments have no impact on the functionality of a program whatsoever and are there for the convenience of the programmer. They are essentially used as small notes so that a programmer can keep track of what is going on in any part of a program and are especially useful with large and complex programs.

6) (**D**) At this point, memorizing the `main` method declaration because it shows up in almost every program and we will understand its components soon, but, for now, know it by heart.

7) (**C**) Choices A, B, and D are incorrect because a `double` value (21.7) is being assigned to an `int` variable which causes an error. However, Choice C works because the `(int)` cast converts the `double` value to an `int` value so that it can be assigned.

8) (**B**) In order to do this question correctly, you must follow the order of operations. Therefore, you complete the division, multiplication, and modulus first going from left to right and are left with `2 - 8 + 2`. This yields a final answer of `-4`.

9) (**C**) Just like the previous questions, you must correctly utilize the order of operations. Based on that, after completing the multiplication and division first, you are left with `1 + 6 - 4`. Remember that `5 / 3` returns `1` because you are dividing integers so the remainder is truncated. Then, you end up with a final answer of `3`.

10) (**A**) In this question, the focus of the code given is on the `4` lines in the body of the `main` method. We must evaluate the expressions that determine the values for the `result1` and `result2` variables to determine what numbers are printed out. We can achieve this using the order of operations. For `result1`, you complete the division, multiplication, and modulus first going from left to right and are left with `2 - 8 + 2`. This yields a final answer of `-4`. For `result2`, the parantheses take precedence over everything else so you get `-4 / 3 * 4` which returns `-4`. Remember that `-4 / 3` returns `-1` because you are dividing integers so the remainder is truncated.

11) (**D**) I and II are correct because `x` is being assigned an `int` value and that matches what is being declared. III is incorrect because a `double` value is assigned when an `int` is declared. IV is incorrect for the very same reason as a `String` value is being assigned to a `double` variable. V is correct because `x` is a `String` variable and is given a `String` value.

12) (**A**) Since an equality operator (a type of relational operator) is being used, we know that `true` or `false` will be returned. `654321` and `654321.0` represent the same numerical value which is what the `==` operator so `true`is returned.

13) (**B**) For the variables, `x` has a value of `true` and `y` has a value of `false`. `z` has a value that this opposite of the value of `x`because of the `!` operator so it stores `false`. The first result is based on the expression `(x | y)` which is equivalent to `true OR false`. With the OR operator, if either of the values is `true`, the expression returns `true`, so result 1 is `true`. For result 2, the expression is `(y & z)` which translates to `false AND false`. With the AND operator, if either of the values is `false`, `false` is returned, making result 2 false. For our final result, `!z` is the experssion and can also be thought

of as `NOT false`. With the NOT operator, it returns the opposite of the original value so result 3 is `true`.

14) (**A**) First of all, no error will occur in this code segment as everything is properly declared, assigned, and evaluated. Since `big` is 45.67 and `small` is 45, we can go ahead and evaluate the expresion that determines the value fo the `result` and, hence, what is printed out. The boolean expression is `(big > small && small != 100)`. Both halves of this expression, `(big > small)` and `(small != 100)`, need to be `true` for this expression to evealuate to `true` as we are working with the `&&` operator. We know that `(big > small)` is `true` since 45.67 is greater than 45 and `(small != 100)` is `true` as well as 45 does not equal 100. Therefore, the variable `result` gets a value of `true`.

15) (**A**) For Choice A, the declared `String` value does not match the `5` that is assigned resulting in an unsuccessful compilation. For the remaining choices, there's no such error. For Choice D, `"5"` is considered to be a `String` value.

16) (**D**) Getting this question correct is entirely dependent on your understanding of `++` and `--` operators. Remember that when these operators are used, if they are before their respective variables, their values are changed before the expression is evaluated. However, if they are after, the values are also increased or decreased after the expression is evaluated. In this case, `a = --b * c++;` is basically `a = 4 * 3` which is 12 and `b` and `c` both finish with a value of 4.

17) (**C**) Getting this question correct is entirely dependent on your understanding of `++` and `--` operators. Remember that when these operators are used, if they are before their respective variables, their values are changed before the expression is evaluated. However, if they are after, the values are also increased or decreased after the expression is evaluated. In this case, `(++g * 8)` is basically `(4 * 8)` which is 32.

18) (**B**) Getting this question correct is entirely dependent on your understanding of `++` and `--` operators. Remember that when these operators are used, if they are before their respective variables, their values are changed before the expression is evaluated. However, if they are after, the values are also increased or decreased after the expression is evaluated. In this case, `(g++ * 8)` is basically `(3 * 8)` which is 24.

19) (**D**) Getting this question correct is entirely dependent on your understanding of `++` and `--` operators. Remember that when these operators are used, if they are before their respective variables, their values are changed before the expression is evaluated. However, if they are after, the values are also increased or decreased after the expression is evaluated. In this case, `a = b-- * c++;` is basically `a = 5 * 3` which is 15 and `b` and `c` both finish with a value of 4.

20) (**C**) Choice C is invalid because for the variable declaration, the variable type is not given, resulting in a compilation error. The rest of the choices are properly assigned and declared.