CHAPTER 1

Introduction

In my experience, learning how to program (in typical computer science classes) can be very difficult. The curriculum tends to be boring, abstract, and unattached to "real world" coding. Owing to how fast technology progresses, computer science classes tend to teach material that is very quickly out of date and out of touch. I believe that teaching programming could be much simpler, and I hope this book achieves that goal.

Note There's going to be a lot of tongue-in-cheek humor throughout this book, but this first part is serious. Don't worry, it gets better.

1.1 Problem Solving

Before you learn to program, the task can seem rather daunting, much like looking at a redwood tree on the cover of a book before you climb it. However, over time, you will realize that programming is really about problem-solving.

On your journey toward learning to code, as with so much in life, you will encounter many obstacles. You may have heard it before, but it really is true: the path to success is to try, try, and try again. People who persevere the most tend to be the most successful people.

Programming is fraught with trial and error. Although things will get easier over time, you'll never be right all the time. So, much as with most things in life, you must be patient, diligent, and curious, to be successful.

1.2 About This Book

This book is organized into several chapters, beginning with the most basic concepts. If you already understand a concept, you can safely move ahead to the next chapter. Although this book concentrates on Java, it also refers to other languages, such as Groovy, Scala, and JavaScript, so you will gain a deeper understanding of concepts common to all programming languages.

A. L. Davis, *Modern Programming Made Easy*, DOI 10.1007/978-1-4842-2490-8_1

Tips Text styed like this provides additional information that you may find helpful.

Info Text styled this way usually refers the curious reader to additional information.

Warnings Text such as this cautions the wary reader. Many have fallen along the path of computer programming.

Exercises This is an exercise. You shouldn't see too many of these.

CHAPTER 2

Software to Install

Before you begin to program, you must install some basic tools.

2.1 Java/Groovy

For Java and Groovy, you will have to install the following:

- JDK (Java Development Kit), such as JDK 8
- IDE (Integrated Development Environment), such as NetBeans 8
- Groovy: A dynamic language similar to Java that runs on the JVM (Java Virtual Machine)

Install Java and NetBeans 8 Download and install the Java JDK 8 with NetBeans.[1] Open NetBeans and select File ➤ New Project… ➤ Java Application

Install Groovy Go and install Groovy[2].

2.1.1 Trying It Out

After installing Groovy, you should use it to try coding. Open a command prompt, type `groovyConsole`, and hit Enter to begin.

In `groovyConsole`, type the following and then hit Ctrl+r to run the code.

```
print “hello”
```

[1]www.oracle.com/technetwork/java/javase/downloads/index.html.
[2]http://groovy.codehaus.org/.

Because most Java code is valid Groovy code, you should keep the Groovy console open and use it to try out all of the examples from this book.

You can also easily try out Scala and JavaScript in the following ways:

- For JavaScript (JS), just open your web browser and go to `jsfiddle.net`.[3]
- For Scala, type "scala" in your command prompt or terminal.

2.2 Others

Once you have the preceding installed, you should eventually install the following:

- Scala[4]: An object-oriented language built on the JVM
- Git[5]: A version control program
- Maven[6]: A modular build tool

Go ahead and install these, if you're in the mood. I'll wait.

2.3 Code on GitHub

A lot of the code from this book is available on `github.com/modernprog`.[7] You can go there at any time, to follow along with the book.

[3]`http://jsfiddle.net/`.
[4]`www.scala-lang.org/`.
[5]`http://git-scm.com/`.
[6]`https://maven.apache.org/`.
[7]`https://github.com/modernprog`.

CHAPTER 3

The Basics

In this chapter, I'll cover the basic syntax of Java and similar languages.

3.1 Coding Terms

Source file refers to human-readable code. *Binary file* refers to computer-readable code (the compiled code).

In Java, the source files end with `.java`, and binary files end with `.class` (also called class files). You *compile* source files using a *compiler,* which gives you binary files.

In Java, the compiler is called `javac`; in Groovy it is `groovyc`; and it is `scalac` in Scala. (See a trend here?)

However, some languages, such as JavaScript, don't have to be compiled. These are called *interpreted languages*.

3.2 Primitives and Reference

Primitive types in Java refer to different ways to store numbers and have historical but also practical significance, for example:

- `char`: A single character, such as A (the letter *A*)
- `byte`: A number from -128 to 127 (8 bits[1]). Typically, a way to store or transmit raw data
- `short`: A 16-bit signed integer. It has a maximum of about 32,000.
- `int`: A 32-bit signed integer. Its maximum is about 2 to the 31st power.
- `long`: A 64-bit signed integer. Maximum of 2 to the 63rd power
- `float`: A 32-bit floating point number. This is an imprecise value that is used for things such as simulations.

[1]A bit is the smallest possible amount of information. It corresponds to a 1 or 0.

- `double`: Like `float` but with 64-bit
- `boolean`: Has only two possible values: `true` and `false` (much like 1 bit)

See Java Tutorial—Data Types[2] for more information.

GROOVY, SCALA, AND JAVASCRIPT

Groovy types are much the same as Java's. In Scala, everything is an object, so primitives don't exist. However, they are replaced with corresponding *value types* (`Int`, `Long`, etc.). JavaScript has only one type of number, `Number`, which is similar to Java's `float`.

Every other type of variable in Java is a *reference*. It points to some object in memory. You can think of this as an address.

3.3 Strings/Declarations

A *string* is a list of characters (text). It is a very useful built-in class in Java (and most languages). To define a string, you simply surround some text in quotes. For example:

```
String hello = "Hello World!";
```

Here the variable `hello` is assigned the string `"Hello World!"`.

In Java, you must put the type of the variable in the declaration. That's why the first word above is `String`.

In Groovy and JavaScript, strings can also be surrounded by single quotes (`'hello'`). Also, declaring variables is different in each language. Groovy allows you to use the keyword `def`, while JavaScript and Scala use `var`. For example:

```
def hello = "Hello Groovy!" //groovy
var hello = "Hello Scala/JS!" //Scala or JS
```

[2] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

3.4 Statements

Every statement in Java must end in a semicolon (`;`). In many other languages, such as Scala, Groovy, and JavaScript, the semicolon is optional, but in Java, it is necessary. Much as how periods at the end of each sentence help you to understand the written word, the semicolon helps the compiler understand the code.

By convention, we usually put each statement on its own line, but this is not required, as long as semicolons are used to separate each statement.

3.5 Assignment

Assignment is an extremely important concept to understand, but it can be difficult for beginners. However, once you understand it, you will forget how hard it was to learn.

Let's start with a metaphor. Imagine you want to hide something valuable, such as a gold coin. You put it in a safe place and write the address on a piece of paper. This paper is like a reference to the gold. You can pass it around and even make copies of it, but the gold remains in the same place and does not get copied. On the other hand, anyone with the reference to the gold can get to it. This is how a *reference variable* works.

Let's look at an example.

```
String gold = "Au";
String a = gold;
String b = a;
b = "Br";
```

After running the preceding code, `gold` and `a` refer to the string `"Au"`, while `b` refers to `"Br"`.

3.6 Class and Object

A *class* is the basic building block of code in object-oriented languages. A class typically defines state and behavior. The following class is named `SmallClass`:

```
package com.example.mpme;
public class  SmallClass  {
}
```

Class names always begin in an uppercase letter in Java. It's common practice to use CamelCase to construct the names. This means that instead of using spaces (or anything else) to separate words, we uppercase the first letter of each word.

The first line is the package of the class. A package is like a directory on the file system. In fact, in Java, the package must actually match the path to the Java source file. So, the preceding class would be located in the path `com/example/mpme/` in the source file system.

An *object* is an instance of a class in memory. Because a class can have multiple values within it, an instance of a class will store those values.

Create a Class

- Open your IDE (NetBeans).
- Right-click your Java project and choose New ➤ Java Class.

3.6.1 Properties and Methods

Next you might want to add some properties and methods to your class. A *property* is a value associated with a particular object. A *method* is a block of code on a class.

```
package  com.example.mpme;
public  class  SmallClass  {
    String name;
    String getName() {return  name;}
    void print() {System.out.println(name);}
}
```

In the preceding code, name is a property and getName and print are methods.

3.6.2 Groovy Classes

Groovy is extremely similar to Java but always defaults to public.

```
package com.example.mpme;
class SmallClass {
    String name //property
    def print() { println(name) } //method
}
```

Groovy also automatically gives you “getter” and “setter” methods for properties, so writing the getName method would have been redundant.

3.6.3 JavaScript Prototypes

Although JavaScript has objects, it doesn’t have a class keyword. Instead, it uses a concept called prototype. For example, creating a class looks like the following:

```
function SmallClass() {}
SmallClass.prototype.name = "name"
SmallClass.prototype.print = function() { print(this.name) }
```

Here name is a property and print is a method.

3.6.4 Scala Classes

Scala has a very concise syntax, which puts the properties of a class in parentheses. For example:

```
class SmallClass(var name:String) {
    def print = println(name)
}
```

3.6.5 Creating a New Object

In all four languages, creating a new object uses the `new` keyword. For example:

```
sc = new SmallClass();
```

3.7 Comments

As a human, it is sometimes useful for you to leave notes in your source code for other humans—and even for yourself, later. We call these notes *comments*. You write comments thus:

```
String gold = "Au"; // this is a comment
String a = gold; // a is now "Au"
String b = a; // b is now  "Au"
b = "Br";
/* b is now "Br".
   this is still a comment */
```

Those last two lines demonstrate multiline comments. So, in summary,

- Two forward slashes denote the start of a single-line comment.
- Slash-asterisk marks the beginning of a multiline comment.
- Asterisk-slash marks the end of a multiline comment.

Comments are the same in all languages covered in this book.

3.8 Summary

In this chapter, you learned the basic concepts of programming.

- Compiling source files into binary files
- How objects are instances of classes
- Primitive types, references, and strings
- Variable assignment
- How source code comments work

PART II

Glorified Calculator

Math is the most basic operation a computer can perform. In fact, in the early days of computers, it was the only thing they could do. A computer was basically a glorified calculator. So, it makes sense then that one of the first things a programmer learns is math.

However, unlike most books, this one is going to teach using interesting concepts. Instead of employing abstract or boring concepts, I'm going to talk about zombies, vampires, and various other lethal monsters.

Figure 1. *Dragon*

CHAPTER 4

Math

(Or *Maths*, if you prefer.)

4.1 Adding, Subtracting, Etc.

Your friend Bob was just bitten by a zombie but escaped alive. Unfortunately, there is now one more zombie to worry about.

```
zombies = zombies + 1;
```

There's a shorter way to write the same thing (and we are pressed for time here; the zombies are coming).

```
zombies += 1;
```

Actually, there's an even shorter way to write this, and it's call the *increment operator*.

```
zombies++;
```

Luckily, there's also a *decrement operator* (to use when we kill a zombie).

```
zombie--;
```

Adding and subtracting are easy enough, but what about their cousins, multiplying and dividing? Luckily these symbols are the same in virtually every programming language: * and /.

```
int legs = zombies * 2;
int halfZombies = zombies / 2;
```

Numbers written in Java are of type `int` by default. But what if we want to deal with fractions (such as one-third)?

```
float oneThirdZombies = zombies / 3.0f;
```

No, `3.0f` is not a typo. The `f` makes 3 a `float`. You can use lower- or uppercase letters (`D` means double; `F` means float; and `L` means long).

This is where math starts to get tricky. To engage *float division* (remember, `float` is an imprecise number), we need 3 to be a `float`. If we instead wrote `zombies / 3`, this would result in *integer division*, and the remainder would be lost. For example, `33 / 3` is 10.

MODULO

You don't really need to understand Modulo, but if you want to, keep reading. Imagine that you and three buddies want to attack a group of zombies. You have to know how many each of you has to kill, so that each of you kills an equal number of zombies. For this you do integer division.

```
int numberToKill = zombies / 4;
```

But you want to know how many will be left over. For this, you require *modulo* (%):

```
int leftOverZombies = zombies % 4;
```

This gives you the *remainder* of dividing zombies by four.

4.2 More Complex Math

If you want to do anything other than add, subtract, multiply, divide, and modulo, you will have to use the `java.util.Math` class.

Let's say you want to raise a number to the power of 2. For example, if you want to estimate the exponentially increasing number of zombies, as follows:

```
double nextYearEstimate = Math.pow(numberOfZombies, 2.0d);
```

This type of method is called a *static method.* (Don't worry, you'll learn more about this later.) Here's a summary of the most commonly used methods in `java.util.Math`.

- `abs`: Returns the absolute value of a value
- `min`: The minimum of two numbers
- `max`: The maximum of two numbers
- `pow`: Returns the value of the first argument raised to the power of the second argument
- `sqrt`: Returns the correctly rounded positive square root of a double value
- `cos`: Returns the trigonometric cosine of an angle

- `sin`: Returns the trigonometric sine of an angle
- `tan`: Returns the trigonometric tangent of an angle

For a list of all the methods in Math, see the Java docs.[1]

Sine If you're unfamiliar with sine and cosine, they are very useful whenever you want to draw a circle, for example. If you're on your computer right now, and want to learn more about sine and cosine, please look at this animation[2] referenced in the footnote at the end of this page and keep watching it until you understand the sine wave.

4.3 Random Numbers

The easiest way to create a random number is to use the `Math.random()` method.

The `random()` method returns a double value greater than or equal to zero and less than one.

For example, to simulate a roll of the dice (to determine who gets to deal with the next wave of zombies), use the following:

```java
int roll = (int) (Math.random() * 6);
```

This would result in a random number from 0 to 5.

JavaScript also has a `Math.random()` method. For example, to get a random integer between `min` (included) and `max` (excluded) you would do the following:

```javascript
Math.floor(Math.random() * (max - min)) + min;
```

However, if you want to create lots of random numbers in Java, it's better to use the `java.util.Random` class instead. It has several different methods for creating random numbers, including:

- `nextInt(int n)`: A random number from 0 to `n` (not including `n`).
- `nextInt()`: A random number uniformly distributed across all possible `int` values
- `nextLong()`: Same as `nextInt()` but for `long`
- `nextFloat()`: Same as `nextInt()` but for `float`

[1]http://docs.oracle.com/javase/6/docs/api/java/lang/Math.html.
[2]https://upload.wikimedia.org/wikipedia/commons/0/08/Sine_curve_drawing_animation.gif.

- nextDouble(): Same as nextInt() but for double
- nextBoolean(): True or false
- nextBytes(byte[] bytes): Fills the given byte array with r andom bytes

You must first create a new Random object, then you can use it to create random numbers, as follows:

```java
Random randy = new Random();
int roll6 = randy.nextInt(6) + 1; // 1 to 6
int roll12 = randy.nextInt(12) + 1; // 1 to 12
```

Now you can create random numbers and do math with them. Hurray!

Seeds If you create a Random with a seed (e.g., new Random(1234)), it will always generate the same sequence of random numbers when given the same seed.

4.4 Summary

In this chapter, you learned how to program math, such as:

- How to add, subtract, multiply, divide, and modulo
- Use the Math library in Java
- Create random numbers