Learn the Java Programming Language

This portion of my site is dedicated to teach the basics of the Java programming language. It is geared towards non-programmers so if you already have some programming experience, you should be able to skip some sections. These articles will not go too in depth but will cover enough to help you understand Java code and how it is written. This is intended to teach enough Java to be able to understand the content of my book Building Minecraft Server Modifications. Therefore, I will not reiterate topics that are also covered in the book. If you are looking to simply get an introduction to programming/Java by making some cool Bukkit plugins then this will be a good start. If you are looking to learn how to create executable programs then I suggest taking an Intro to Java course or finding something that is a bit more thorough.

The text is split into the various chapters/sections below. Be sure that you understand a section before moving on to the next. Once you have read the numbered sections you should be able to get started with the first few chapters of my book. Then, as you need to, refer back to some of the additional sections. If you have any questions, feel free to Contact Me.

1.) Intro to Object-Oriented Programming

Java is an object-oriented language. This means the software that you write in it is essentially constructed of different objects. These objects are in the form of classes. Classes contain variables which hold the information of the object, and methods which execute different tasks for the object. Below is a small example of a simple Student class written in Java. We will go over each section of this code, so at this point, you are not expected to understand any of it. This is simply to show you some Java code that you will eventually be able to read.

```
/**
 * A Student contains the name of the student and their gpa
 *
 * @author Codisimus
 */
public class Student {
    public String name; //The name of the Student
    private double average = 100; //The Student's average grade

    /**
     * Constructs a new Student with the given name
     *
     * @param name The name of the Student
     * @return The newly created Student Object
     */
    public Student(String name) {
        this.name = name;
    }

    /**
     * Sets the average including the new grade
     *
     * @param grade The new grade to be averaged in
     */
    public void addGrade(double grade) {
        double sum = average + grade;
        average = sum / 2;
    }

    /**
     * Returns the grade based on the given test scores
     *
     * @return The Student's average
     */
    public double getAverage() {
        return average;
    }
}
```

You may be able to understand some of this code without any prior knowledge. The comments (shown in gray) will help explain what the code does. After taking a look at the code above, it is time to learn the Java syntax so that you may begin reading it.

2.) Syntax

Anatomy of a Class

Before understanding what code means, it is a good idea to know how the code is structured. This will help you break down the code that you are looking at into parts that you can understand. Let's look at that Student class again.

```
/**
 * A Student contains the name of the student and their gpa
 *
 * @author Codisimus
 */
public class Student {
    public String name; //The name of the Student
    private double average = 100; //The Student's average grade

    /**
     * Constructs a new Student with the given name
     *
     * @param name The name of the Student
     * @return The newly created Student Object
     */
    public Student(String name) {
        this.name = name;
    }

    /**
     * Sets the average including the new grade
     *
     * @param grade The new grade to be averaged in
     */
    public void addGrade(double grade) {
        double sum = average + grade;
        average = sum / 2;
    }

    /**
     * Returns the grade based on the given test scores
     *
     * @return The Student's average
     */
    public double getAverage() {
        return average;
    }
}
```

Most of your code will be inside a class such as this. There are several blocks of code in this example. They are determined by the curly braces {}. Each open brace will also have a closing brace to match it. Take note of where the first and last braces are. Everything that is in between the two braces is indented to show that it is within the block of code. The comments, in gray, above the block of code describe what that code does. Blocks of code can also be nested, or within, other blocks of code. Can you figure out how many blocks of code are present in this Student class? The answer is shown on the next page.

```java
/**
 * A Student contains the name of the student and their gpa
 *
 * @author Codisimus
 */
public class Student {
    public String name; //The name of the Student
    private double average = 100; //The Student's average grade

    /**
     * Constructs a new Student with the given name
     *
     * @param name The name of the Student
     * @return The newly created Student Object
     */
    public Student(String name) {
        this.name = name;
    }

    /**
     * Sets the average including the new grade
     *
     * @param grade The new grade to be averaged in
     */
    public void addGrade(double grade) {
        double sum = average + grade;
        average = sum / 2;
    }

    /**
     * Returns the grade based on the given test scores
     *
     * @return The Student's average
     */
    public double getAverage() {
        return average;
    }
}
```

If you matched up the curly braces you would've discovered 4 blocks of code; 1 class + 3 methods. Let's look at each of them individually.

```java
public class Student {

    //Code belonging to the Student class goes in here

}
```

Our Student class has both variables and methods inside of it. Variables are typically listed before methods and tend to take up one line of code each. Methods are larger and have code blocks of their own, indicated by the curly braces.

```java
    public String name; //The name of the Student
    private double average = 100; //The Student's average grade
```

These two lines of code are both variables that belong to the Student class. That is, a student has a name, and an average (GPA).

```java
/**
 * Sets the average including the new grade
 *
 * @param grade The new grade to be averaged in
 */
public void addGrade(double grade) {
    double sum = average + grade;
    average = sum / 2;
}

/**
 * Returns the grade based on the given test scores
 *
 * @return The Student's average
 */
public double getAverage() {
    return average;
}
```

The two blocks of code above are both methods. Methods can be called to complete various tasks. In this case, they are adding a grade for the student and getting the students average.

```java
/**
 * Constructs a new Student with the given name
 *
 * @param name The name of the Student
 * @return The newly created Student Object
 */
public Student(String name) {
    this.name = name;
}
```

This is another method but it is different than the rest. This is known as a constructor. It is used the construct, or create, a new Student object. Constructors are typically placed above all other methods.

This entire class would be in a file named **Student.java**. Above this code there would also be the package declaration and any needed imports. These are important but you do not need to know about them at this point. Just know that they will be located outside the class's code block.

Comments

Comments are added to the code to help explain it to anyone who may be reading it. You can see several examples of each in the code above. A good programmer will always comment their code because they realize that it will help them (and others) understand it in the future. Comments will not affect how the code is executed in any way. There are several different formats for adding comments.

- Comments found between /** and */ are used for documentation. They can span over several lines of code.
- Comments found between /* and */ are similar but are used for simple text rather than documentation. The can span over several lines of code as well. These may be used for long notes or to comment out an entire block of code so that it does not execute.
- // is used when you want to add comments for a single line. Everything to the right will be marked as a comment.

Primitive Data Types

The objects that you work with in Java have data stored within them. This data is stored in the variables. For example, our student object stores its name and average. There are various types of data that can be stored. The most primitive types that you will encounter are **int**, **double**, **boolean**, **char**, and **long**. There are more, but they are less common.

- **int**, **double**, and **long** are all used to store numbers.
- An **int** represents an integer between -2,147,483,648 and 2,147,483,647.
- A **long** may be used for an even larger range. Usually, an **int** will be large enough to store any number that you will encounter. One occasion which would require a **long** is when dealing with time in the form of milliseconds which is done quite often. A month is 2,678,400,000 milliseconds which is not within the range of an **int**. Therefore it is represented with a **long**.
- A **double** is used when you want to have a number that includes decimal places.
- A **boolean** can only represent 1 of 2 states; *true* or *false*. Sometimes it is more understandable to think of the values as *on* and *off* or *yes* and *no*.
- A **char** is a single character. This may be a letter, number, or symbol such as *X*, *8*, or *§*.

Additional Data Types

You can store more than just the data types listed above. There are numerous classes in Java that are at your disposal. One that you will use a lot is the **String** class. This is a sequence (or string) of characters. Not only may it be used to store words, it can also store sentences, paragraphs, and as much text as you want it to. Strings may be created by enclosing text between quotations as shown below.

```
"This is a String!"
```

You may also want to have a collection of data. For example, if we were to build upon our Student class, a student should store a collection of grades (of type **double**) rather than a single average. Java has many forms of collections to suit your needs. One that you will often use is a **List**. These collections will be explained in further detail later so that you know which kind to use at which time.

An object may also store another object. For example, we may have programmed a separate class called Course. A course object would have a specific subject (String) such as `"Math"` or `"Science"`. A student may store a course, or collection of courses, that they are enrolled in.

Operators

There are several operators that are used in the Java language. Most of these will be explained throughout other sections. However, we will introduce some here.

==	is used to see if two values are equal. Similarly, != checks for the opposite (not equal).
<, >, <=, >=	are also available for comparing values such as numbers.
=	is used to assign a value to a variable, such as assigning the value of the average variable in the student class above.
+, -, *, /	are mathematical operators that you are probably familiar with (addition, subtraction, multiplication, division).
%	is a useful operator that you may find yourself using someday. It gives you the remainder of a division. For example, 10 % 3 = 1 and 123 % 100 = 23.

Other important operators will be explained as they are introduced in context.

Modifiers

Modifiers are used in declaring classes, methods, and variables. Two modifies that you see in the code above is public and private. This determines which classes/methods have access to the class/method/variable. This will be further explained in the methods and variable sections.

The Code

The bulk of code that is written is within methods. When you program your methods, you assign variables, call other methods, and more. Look at the methods from the Student class for examples.

Each line of code will end in a semicolon (;) which indicates the end of the statement. Of course there are exceptions to this such as lines which start a new block of code. Comments also need not end with a semicolon. Sometimes, long lines of code will be broken up to span multiple lines. To view the entire statement, look for where the semicolon is placed.

Much more will be explained regarding the contents of code in the coming sections.

Now you should understand most of the code provided in the Student class. Next we will learn more about variables and how to declare them.

3.) Variables

It was mentioned that variables store data in a Java program. In this section, you will learn the basics of declaring variables and how to assign values to them.

Declaring Variables

Declaring a variable can be very simple. When doing so, you must provide the **type** of the variable and a **name** for it. The **type** refers to the data type such as int, double, boolean, String, and so on. In Java, each variable is declared as a specific type so that there is no type mismatch. You won't expect a value to be an int when it is actually a String. The **name** of the variable can be pretty much anything you want except for keywords such as int, public, class, and so on. The name must not have any spaces but can be multiple words mashed together. It is typical to use camel casing in this scenario. Camel casing is a form of capitalization which is named be the way the word looks. It only applies when there is more than one word in the name. The first character would be lowercase while each remaining word has its first letter capitalized. You will understand when you see the examples below.

```
thisIsCamelCasing

soIsThis
```

Apple is well known for using camel casing in their product names.

```
iPod, iMac, iPhone, iPad
```

Variables may be as short or as long as you want. Some programmers would prefer 1 letter names (such as `i`) because they are more compact. Others prefer longer names (such as `gradePointAverage`) which explain the variable's purpose. Either is fine, as long as you can understand the code. You may include numbers and symbols in the name but it should begin with a letter.

The type and name are written in that order and then the line is completed with a semicolon. Below is the simplest form of declaring a variable.

```
int gradePointAverage;
```

There are three places where you will declare variables.

1.) Inside a method block. These are called local variables because only that method has access to them.
2.) Within a method header. These are known as parameters and will be discussed in the next section. They function similarly to variables declared within a method block.
3.) In a class yet separate from the method block. These are known as fields. They may be accessed by any method within the class.

Modifiers

public/private

Modifiers were briefly mentioned in the Syntax section. They may be used when declaring fields (variables which are declared outside the class methods) to determine the access of the variable. There are 2 modifiers which you will often see; private and public. Without a modifier, only the class that holds the variable, and other classes that are in the same package can access the variable. The private modifier limits the access to only the declaring class. The public modifier expands the access to allow any other class to read and modify it. These modifiers are used to prevent other programs from modifying your data in an incorrect manner. To be secure, it is best to make all of your variables private except for special circumstances.

static

The keyword static means that there will only be one instance of the variable. This means that objects from the same class will 'share' the variable. For example, a Student class may have a static variable name `schoolName`. Each student that goes to the school will have the same value for `schoolName` therefore it should be static. Alternatively, a variable of `studentName` cannot be static because each student has their own name. Static is another modifier which only applies to fields. This is further explained in my book when you are expected to use static variables.

final

Declaring a variable as final means that the value of the variable will not change. This is normally used for constants such as the amount of hours in a day. When declaring a variable as final, the naming conventions change. You no longer use camel casing. Instead you use full uppercase and separate words with an underscore. An example of this would be `HOURS_PER_DAY`.

These three optional modifiers should be added in the given order. Therefore, if you were to add all three, it would be as follows:

```
private static final int HOURS_PER_DAY;
```

Instantiating a Variable

Declaring a variable is only half the process, you must also instantiate it. Instantiating the variable gives a value to it. Otherwise, the value would be its default initialization depending on the type. For example, an int, double, or any other number is initialized to 0 (zero) when it is declared. A boolean is initialized to false. Also any object, such as a String, is initialized to null meaning that no value yet exists.

Instantiating a variable is simple and can be done on the same line that it is declared. In order to indicate that you are setting the value, you simply need to add = and then the value that you wish to store inside of the variable. Examples of this are found in the Student class in the previous section and in the line of code below.

```
private static final int HOURS_PER_DAY = 24;
```

Remember that Strings must be enclosed in quotations as shown in the following example.

```
public name = "Cody Sommer";
```

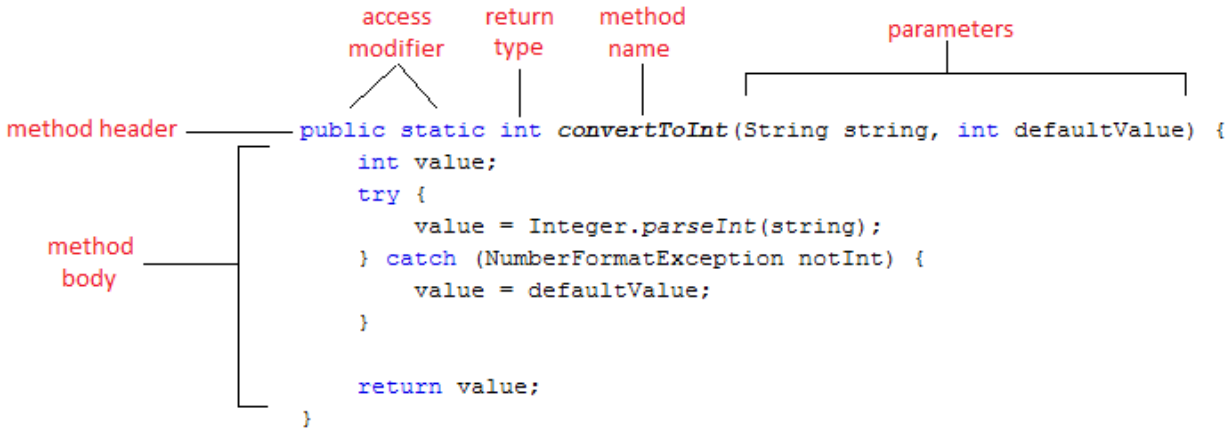
For final variables, this must be done at the time of declaration or within the constructor. For other variables, this may be done at any time.

Next we will discuss methods and how declaring them is similar to declaring variables. We will also go over how variables should be declared inside a method.

4.) Methods

Method Headers

A class can have any amount of methods within it. Declaring a method is much like declaring a variable in that you include modifiers, data type, and a name in the method header. Below is a sample method that has each part labeled.



The data type in a method header defines the type of data that is returned from the method. For example, the **`convertToInt`** method above returns an int. If a method does not return a value then the keyword `void` is set as the return type.

Parameters

The last part of the method header is the parameters. Parameters are variables which are passed to a method. Parentheses are placed after the method name. Within them, you declare each variable which must be passed to the method. You may add as many parameters as you need. The example method above requires two parameters. The first is a String which will be converted to an int such as "100". The second value is the default int value to return in case the String which was passed is not actually an int. For example, all of the following are Strings that will not be successfully parsed to an int value.

- `"one hundred"` (Only numerals can be parsed)
- `"100.0"` (Integers do not have decimal places)
- `"  100 "` (there are extra spaces in this String)
- `"hello world!"` (This is not even a number)

Method Body

The body of the method is where its code is located. Within here you may also declare more variables as shown above. These are known as local variables. Both local variables and parameters are not accessible by any other method or class.

Within the method body you write the code to execute the method. In this case, we attempt to convert the String value into an int. Remember that you may access class fields (variables declared within the class yet outside the method) from inside the method.

If the return type is not void, you must have a return statement within your method body. This statement will be the final line that is executed in the method so it is usually on the last line. Notice above that we return **value** which is an int.

Even if the return type is void, you may still exit the method using `return`. You simply don't return a value. Therefore the code would be:

```
return;
```

Calling a Method

In order to call a method, you reference it by its name and then pass the needed parameters to it. For example, the following line of code would call the above method.

```
convertToInt("23", 0);
```

If the method has no parameters then you leave the parenthesis empty like so:

```
convertToInt();
```

This line of code may be called from another method or another class. The convertToInt method returns an int so you would want to assign the value to a variable.

```
int i = convertToInt("23", 0);
```

You now know the very basics of programming in Java. It should be enough to get you started with my book, Building Minecraft Server Modifications. If you feel that something needs further explanation then please Contact Me about it. I will continue to provide more sections regarding various aspects of the Java language.