Java

Neil Kingdom

June 30, 2024

Contents

1 Abstract

I intend for this document to differ from the rest. I think Java actually is a great first language to learn. Most junior programmers will suggest learning Python first, but I think that this couldn't be further from the truth. Python is a fairly simple language, that much is true, but that's what makes it a bad first language to learn. Java is somewhere in the middle. It doesn't force you to manage memory, but it is strongly typed, meaning that you at least learn to use types properly, and it's a good language for learning the OOP paradigm. This document will intentionally be dumbed down compared to my other ones and explain basic concepts at a slower pace, so that this can act as a starting point for new programmers.

2 Introduction

Java is an Object Oriented Programming (OOP) language, which has features like static typing, classes, garbage collection, cross-platform support, and a rich ecosystem of libraries and tools. If none of that made any sense to you, fret not, because I will be explaining each of these things throughout the document. I'm sure most people are familiar with Java, since it became one of the most popular languages around the 2000s era. Java was created in 1995 by Sun Microstems, which was later acquired by Oracle.

3 The Java Virtual Machine

One of the reasons Java was so popular at the time is because programming languages weren't very portable across different platforms such as Linux, Mac, and Windows. You usually had to rewrite portions of code, or even the entire application for each platform that you wanted to support, which cost a lot of development money. Java was one of the first "managed" languages, as opposed to a traditional compiled language. A compiled language uses a compiler to translate the code into executable assembly instructions. The assembly instructions are produced for the specific architecture of the computer's processor e.g. x86 for Intel/AMD or ARM. A managed language solves the issue of portability by having a piece of software run on the system which acts as a middle man. This piece of software for Java is called the Java Virtual Machine, or JVM. Rather than compile each application written in Java into assembly code that runs natively on the user's hardware, why not have a single application i.e. the JVM be compiled for each platform and then have it interpret our Java applications. Java applications are not compiled into assembly directly, but rather into an intermediary language known as byte code. Byte code is specific to Java, and can only be read by the JVM. It's the JVM's job to then take the byte code and give it to the Java Runtime Environment (JRE) to be interpreted and executed. If this is confusing, don't worry too much – all you need to know is that we need the JVM to run Java applications and that it makes them compatible with all architectures/platforms.

4 The Java Development Kit

Typically, managed languages will come with some sort of development kit, which is kind of like a suite of applications bundled into one installer. Usually this is called a Software Development Kit (SDK), but Java calls theirs the Java Virtual Machine (JDK). Non developers actually don't need the entire JDK – they only need the JRE to be able to run Java applications. The JDK includes the JRE, but also adds the JVM, core class libraries, the Java compiler (javac), the Java debugger (jdb), and documentation. The history of the JDK can be a bit confusing because of Sun's acquisition by Oracle. The JDK used to be called J2SE. J2SE still exists nowadays, but has been renamed to Java SE (Standard Edition). There is also Java EE (Enterprise Edition) and Java ME (Micro Edition). Nowadays, these have been delegated to the enterprise domain of Java, which will be beyond the scope of this document. When Sun was acquired by Oracle, they created an open-source reference to J2SE/Java SE called the OpenJDK, which is what we are typically referring to when we say JDK. The OpenJDK is under the GPL-2.0 license. That's probably gobblediegook to you, but it's essentially just a license which states that the software can be distributed freely and modified without legal reprecussions. The OpenJDK reference began after Java 7 i.e. Java 8.

You might hear about Java 8 more frequently than the other releases, and that's because of this change. All versions of Java above 8 are backwards compatible with previous releases down to and including 8, but are not necessarily backwards compatible with 7 and below. Since the OpenJDK is just a reference guide, the actual implementation details are up to whichever vendor wants to do the implementing. For example, Eclipse Adoptium is a working group who have an implementation of the OpenJDK which you can download, or you can use Oracle's own implementation of the OpenJDK. IBM has their own, Jetbrains has their own, Red Had has their own, etc. They should all function the same in your case, but I usually like to get it from Oracle.

As a developer, it's also good to be familiar with Long Term Support (LTS) releases. An LTS release is a version of software that an organization plans to support for a long time (bet you didn't see that coming). That means that they continue to release new versions of the software, but that the bug fixes which go into those new releases usually also end up in the LTS versions as well. LTS releases are more stable and are tested more thoroughly, but don't tend to receive new features. LTS releases are generally preferred by businesses and for good reason, but as a developer building your own programs, LTS releases don't matter much. Until current date, the LTS releases for the JDK are 8, 11, 17, and 21.

5 The Java Class Libraries

Recall that I stated the JDK provided the developer with core libraries. This set of core libraries is called the Java Class Libraries (JCL). A library is just a set of prewritten code that we can use within our programs. We can install and use additional libraries beyond what's in the JCL, but the JCL provides developers with most of the basic/core functionality that is expected. When we import a library, we do so with a statement like the following: import java.util.Random;. In java, paths to files are represented using dots rather than slashes, but you can think of this as importing a file called Random from the relative path ./java/util/Random, where the Random file contains all of the library code for performing operations with random numbers. Point being that you can tell this is a part of the JCL due to the fact that the initial directory is the java directory. Extention libraries which are not directly part of the JCL are usually located in the javax directory, which stands for Java extentions.

6 Variables

Finally, after talking about Java's ecosystem for a while, let's now begin to actually discuss the language. For those who are using this document as a starting point, I'll explain the concept of variables, but for others, you can skip the next couple of sections. In mathematics, we use variables such as x to represent some unknown quantity. In programming, we use variables to represent any form of data, though the data does not necessarily need to be numeric like in maths. In programming, the value of a variable is also usually, though not always, known. We tend to avoid using labels like 'x' in programming because we use a lot of variables for a lot of different things. Maths likes to keep things concise, but in programming we can affort to label our variables with more descriptive names, so instead of x, we might use the label 'n', 'num', or 'number'. The analogy of a bucket is usually used for variables. Each variable is like a bucket which contains some data. On a more technical level, variables expand to lookup addresses which the CPU uses to index memory and locate the value stored at that address. Depending upon the type of data that is stored in the variable, we may need to allocate more or less space in memory to be able to store it. For now, memory is not too important though, so just know that variables can store values and that they can be modified or altered during the execution of a program.

7 Data Types

Each variable is prefixed by a data type, which indicates to the compiler the type of data that is expected to be present within the variable. Java splits data types into both primitive types and objects. A primitive

type is essentially a standard type that you'd find in any language, and is a fundamental building block for creating object types. Object types are user-defined types which are composed of multiple items that are either primitives or other objects. Hopefully it will help to see some examples of primitive types and object types to bring things back to ground 0. In Java the primitive types are as follows:

Data Type	Size	Default Value
byte	8 bits / 1 byte	0
short	16 bits / 2 bytes	0
int	32 bits / 4 bytes	0
long	64 bits / 8 bytes	0L
float	32 bits / 4 bytes	0.0f
double	64 bits / 8 bytes	0.0d
char	16 bits / 2 bytes	'\u0000'
String	Depends	Null
boolean	8 bits / 1 byte	false

In Java, byte, short, int, long, float, and double are considered to be numeric types; String and char are considered to be text-based types; and boolean is it's own category of type. Note that Strings are not actually primitive types, but Java treats them as such. Strings are objects, which you can tell by the fact that they begin with a capital letter, unlike the other primitive types. Each of the primitive types do actually have object counterparts. For instance, int has the Integer class, boolean has Boolean, etc. We'll discuss these object counterparts more later. For now, I'll provide a brief use-case for each type.

- **Byte:** The byte type is the smallest integer type that Java provides. As the name would imply, it can store a byte (or 8 bits) of information. Java primitives are only signed values, meaning that they can either store negative or positive numbers. This comes at the cost of reducing the range of values that can be stored. An unsigned byte could store values from 0-255, but Java bytes can only store values within the range -128 to 127. As we'll see, the object counterparts I mentioned earlier will allow us to use unsigned artithmetic.

- **Short:** Short, short for short integer, is rarely used in Java. The byte type actually has some practical purpose since a lot of programming deals with bytes of information, but short is much less common, and is more a relic of the 32 bit past of computing. The short type can accept a range of values from −32768 to 32767.

- **Int:** The int type, short for integer, is the most commonly used integer-based type on 64-bit architectures. There is nothing special about it, but for historical purposes, int was adopted as the sort of standard numerical type in programming. It has a good balance between range of values without taking up too much memory. The range of the int type is −2,147,483,648 to 2,147,483,647.

- **Long:** Long, short for long integer, is the largest integer-based type in Java. It is primarily used when working with very large numbers that exceed the range of an int (which doesn't happen too often). It can store values in the range −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

- **Float:** Float, which is short for "single precision floating point", is the first non-integer based numerical type of the list. A floating point number is essentially a decimal. Even though the float type occupies the same amount of memory as an int, its range of values is lower, since it utilizes most of the bits to represent the decimal portion. Floats utlilize the IEEE-754 standard, which I won't go into here, but is definitely worth learning about if you're not familiar with it. Because of this standard, floats must always be signed. This is the case in all programming languages – floats are never unsigned.

- **Double:** Double, which is short for "double precision floating point", is also a floating point number like float, however it has twice as many bits, hence twice the precision. Once again, doubles adhere to the IEEE-754 standard, and must be signed.

- **Char:** The char type, short for character, represents a single character. Unlike in other languages where characters are 1 byte and can only represent characters from the ASCII standard, Java uses unicode for it's characters, which requires them to occupy 2 bytes. Internally, Java stores unicode characters using the UTF-16 encoding, but this gets converted to the user's default character set, which is usually UTF-8. The benefit of unicode over ASCII is that we can store and represent characters from other languages, and even other glyphs such as emojis. Characters can be non-printable as well, which are usually reserved for special purposes by the OS. A char can store any unicode character within the range \u0000 to \uFFFF, which is a range of 65535 characters.

- **String:** As I mentioned, Strings are not truly primitive types, but they are used so often that Java even states to treat them as primitives. A String is basically just a series of characters packed into one word or sentence or sequence. This is how we primarily store text in Java, as we'll come to see more in the future.

- **Boolean:** The boolean type stores a boolean value, which means a value of either true or false. This is used in conditional statements as we'll come to learn about more.

8 Writing Hello World

We're going to jump the gun a bit and write our first program. It is standard practice to write what's called a hello world program when learning a new language, as sort of a test to ensure everything is working correctly. Let's see what this looks like in Java, and then I can explain some concepts:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

I recall seeing this the for the first time and being a little indimidated by how wordy these statements are. We'll eventually learn what all the keywords mean, and this will begin to make more sense, but for now I want to look at the bigger picture of what's transpiring in this code. Java has a bit of a unique rule that most languages don't have, which is that every file with a .java extension must contain a class, and furthermore, the name of the file must match the name of the class contained within it. In this example, we've created a file called HelloWorld.java, which contains the code in the snippet above. The class is named HelloWorld, using the same casing as the file. In order for any program to run in Java, we need one and only one entry point, which is where the program begins execution. In almost all programming languages, the entry point is a function named main(). In some other programming languages, the declaration of main() is flexible, but in Java it must follow the syntax we've used in the example i.e. public static void main(String[] args). args is a variable, which is the only thing that we're allowed to name differently, though it's good to keep the convension of calling it args. Within the entry point of our program, we use the statement System.out.println(”Hello, World!”); to print out the string ”Hello, World!” to the console.

Writing the program is one thing, but running it is another. The way you execute this code will be dependent upon your setup. There are essentially two options that you have: either execute the code from a terminal, or install an Integrated Development Environment (IDE) to manage compiling and running your code. If you decide to go the IDE route, you can use one of the following for Java: Eclipse, Netbeans, Idea (by IntelliJ), or VS Code. Eclipse, Netbeans, and VS Code are all free, whereas Idea has a trial period, but ultimately requires a subscription. Idea is the best of all of them though. VS Code is good, but I wouldn't recommend it for newcomers, since it requires additional setup. Between Eclipse and Netbeans, I personally prefer Eclipse, but I know a lot of people hate Eclipse and would much prefer Netbeans. I will be explaining the command line so that we can familiarize ourselves with how the build process actually works in Java.

In order to run our code, we first have to compile it. We do this using the java compiler called javac, which is included in the JDK. The Java compiler will take a .java file and convert it into a .class file, which contains the intermediary byte code interpreted by the JVM. Run the following command:

javac HelloWorld.java

This will generate HelloWorld.class. Now to run the program, we use the java command like so:

java HelloWorld

The java command accepts the name of the class in which the main method i.e. our program's entry point exists. This should output the words "Hello, World!" to the console. Compilation becomes a bit more complicated when we introduce multiple .java files, so we'll revisit the build process later on in the document. For now, this process of running javac followed by the java command should suffice.

9 Object-Oriented Programming

As I've alluded to multiple times now, Java is an Object-Oriented Language (OOP). This paradigm is known as imperative programming. OOP languages operate entirely around the concept of objects. An object is composed of two things: state and behaviour. Objects are nice because they couple both state and behavior into one localized construct, whereas in a procedural language, for example, state and behaviour are not tethered to objects, and therefore sort of exist in the ether. I know most of what I'm saying is going to go over most new programmer's heads entirely, and that's fine, but the takeaway is that objects are foundational to the OOP paradigm, and that they can contain both state and behaviour. I might also tack on a note for the mathmeticians, which is that objects are like nodes in a graph. Each node is able to make connections and exchange information i.e. the state with other nodes in the graph. Behaviour defines operations (functions), which operate on the state of other nodes.

What do I actually mean by state and behaviour, because I've been extremely vaccuous and vague. I've done so intentionally, because it's easy to get lost in the details and forget the larger picture, though both are important to grasp. State are essentially our variables that we discussed earlier. If you were take a snapshot of the program or pause execution, state could be defined as the status of all variables contained in one or more object(s). Behaviour defines operations which act upon state. Java uses the term "instance variable" to describe a variable which belongs to an object, though in other OOP languages, these are sometimes refered to as "member variables". Java also uses the term "method" to describe the behaviours of an object. Other programming languages – especially procedural ones, will have functions. A method is a function, but more specifically it refers to a function that belongs to an object. Thus the terms instance variable and method are synonymous with state and behaviour.

10 Objects and Classes

Objects are heavily tied to what we call classes in OOP. A class is the contract or blueprint which constructs an object. In other words, a class is the code which declares the member variables and methods that will belong to the object. The word "instantiation" is an important one in the OOP paradigm. Another way of saying that we create an object from a class is to say that we instantiate the class, or that we make an instance of the class. Effectively, we can replace the word "object" with "instance of the class". Objects are also variables, and so when we instantiate a class, we basically create a new variable whos data type is that of the class. Looking back at our hello world program, recall that we had to make a class called HelloWorld. This class never actually got instantiated because we don't normally instantiate the class which contains the main() method. Also, note I said main() method, because that's what main() is – a method. It is considered part of the behaviour ascribed to our class. Let's look at how we can create a class which contains both instance variables and methods:

```java
public class MyClass {
    // Instance variables
    private int aNumber;
    private String aString;

    // Methods
    public void myMethod1() {
        // TODO
    }

    public void myMethod2() {
        // TODO
    }
}
```

Theres a good amount to disect here. The class we've created is called MyClass, and must exist in a file called MyClass.java. When programming, it's useful to be able to leave messages within the code for other programmers who might be reading it, or to remind yourself of something. These are called comments. Comments in Java begin with two slashes (//). This is not universal in all languages, but must programming languages use two slashes. Anything that comes after the slashes will not be executed, so long as it remains on the same line. We've created two instance variables called aNumber, which is of type int, and aString, which is of type String. We've also declared two methods, but not actually added any implementation details describing what they do yet, hence the TODO comments. The word void also happens to be a special data type, but we'll get into that more in a bit.

11 Methods

Let's get more in depth as to how methods work. You'll notice that most methods begin with the public keyword. This is known as an access modifier, but I'll dedicate a whole section to that, so don't worry about it too much for now. After the access modifier comes what's known as the return type. A return type specifies the data type that we expect the return value to be. Methods can optionally accept inputs and optionally produce outputs. The return value is essentially a singular output. The void type indicates that this particular method does not actually return anything i.e. it does not produce any output. After the return type, we give the method a name. Two methods within the same class cannot share the same name, though you can have two distinct classes that each have a method with the same name. The brackets proceeding the methods name are what's called the parameter list. The parameter list can optionally define multiple variables that will be bound to whatever inputs the method is called with. We'll look at an example of this shortly. Finally, we have the method's body, which is represented with a set of curly braces. The curly braces denote the 'scope' of the method.

Imagine now that you're back in math class. You have a function called f(x). Remember that f(x) produces the output y, so it can also be written as y = f(x). Now lets say f(x) is defined as x + 2. Let's see how we'd write this in code:

```java
public int f(int x) {
    return x + 2;
}
```

Here we create a method called f, which accepts a single input according to the parameter list. Rather than calling x an input, we usually call it a parameter. The parameter x is of type int, so it is expected that we call f() with a number e.g. f(3);. The return keyword specifies the output, or return value of the method. I've

failed to mention up until now that every statement in Java is terminated with a semi-colon. This is how the compiler can know that we want to return x + 2, and not just x, since the statement ends after the 2. To throw more terminology at you, another word for 'calling' a method is to say that we 'invoke' the method. Also, when we invoke a method, the value that we pass in as an input is called an argument. This differs a bit from the parameter. Using these terms, we can say something like the following: "When we invoke f() with the argument 3, the parameter x is assigned the value 3. The method f() returns the sum of the parameter x and the value 2, which yields 5."

12 Creating and Instance of the Class

Recall that the proces of creating an object is also referred to as instantiating the class or creating an instance of the class. The reason that we want to be able to create objects is because methods can only be invoked by an object. A commonly used example is the creation of a dog object. We can say that one of the behaviours of a dog is to speak. Let's create a dog class, alongside a method that describes the speaking behaviour, and then actually invoke the speak() method:

```
public class Dog {
    public void speak(String bark) {
        System.out.println(bark);
    }

    public static void main(String[] args) {
        Dog corgi = new Dog();
        corgi.speak("yip");

        Dog bernese = new Dog();
        bernese.speak("woof");
    }
}
```

When we execute this program, both yip, followed by woof, will be printed to the console. Let's follow the execution path of the program. Keep in mind we always start at main(). The statement Dog corgi = new Dog() is broken into two parts. The first part "Dog corgi" defines a variable called corgi, which is of type Dog. In order to instantiate a class in Java, we need to use the new keyword, followed by some special syntax i.e. Dog() (more on this later). This constructs a unique Dog object. As I mentioned ealier, methods can only be invoked from objects. Now that we have corgi as a Dog object, we invoke the method speak() using the dot operator (.) e.g. corgi.speak(). Since the speak() method expects us to provide it with an argument of type String, we pass in the String "yip". The speak() method binds/assigns this value to the bark parameter, and prints it to the console. The same procedure is repeated for the next instance of Dog, which is bernese, but this time we invoke the speak() method with the argument "woof".

13 Creating a Java Archive

14 Manifest Files

15 Debugging with JDB

16 Ant, Gradle, and Maven