Chapter 1: Getting started with Java Language

Java SE Version	Code Name	End-of-life (free1)	Release Date
Java SE 10 (Early Access)	*None*	future	2018-03-20
Java SE 9	*None*	future	2017-07-27
Java SE 8	Spider	future	2014-03-18
Java SE 7	Dolphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4	Merlin	prior to 2009-11-04	2002-02-06
Java SE 1.3	Kestrel	prior to 2009-11-04	2000-05-08
Java SE 1.2	Playground	prior to 2009-11-04	1998-12-08
Java SE 1.1	*None*	prior to 2009-11-04	1997-02-19
Java SE 1.0	Oak	prior to 2009-11-04	1996-01-21

Section 1.1: Creating Your First Java Program

Create a new file in your text editor or IDE named `HelloWorld.java`. Then paste this code block into the file and save:

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

Run live on Ideone

Note: For Java to recognize this as a **`public class`** (and not throw a compile time error), the filename must be the same as the class name (`HelloWorld` in this example) with a `.java` extension. There should also be a **`public`** access modifier before it.

Naming conventions recommend that Java classes begin with an uppercase character, and be in camel case format (in which the first letter of each word is capitalized). The conventions recommend against underscores (_) and dollar signs ($).

To compile, open a terminal window and navigate to the directory of `HelloWorld.java`:

```
cd /path/to/containing/folder/
```

Note: cd is the terminal command to change directory.

Enter `javac` followed by the file name and extension as follows:

```
$ javac HelloWorld.java
```

It's fairly common to get the error `'javac' is not recognized as an internal or external command, operable program or batch file.` even when you have installed the `JDK` and are able to run the program from `IDE` ex. `eclipse` etc. Since the path is not added to the environment by default.

In case you get this on windows, to resolve, first try browsing to your `javac.exe` path, it's most probably in your `C:\Program Files\Java\jdk(version number)\bin`. Then try running it with below.

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

Previously when we were calling `javac` it was same as above command. Only in that case your `OS` knew where `javac` resided. So let's tell it now, this way you don't have to type the whole path every-time. We would need to add this to our `PATH`

To edit the `PATH` environment variable in Windows XP/Vista/7/8/10:

- Control Panel ⇒ System ⇒ Advanced system settings
- Switch to "Advanced" tab ⇒ Environment Variables
- In "System Variables", scroll down to select "PATH" ⇒ Edit

You cannot undo this so be careful. First copy your existing path to notepad. Then to get the exact PATH to your `javac` browse manually to the folder where `javac` resides and click on the address bar and then copy it. It should look something like `c:\Program Files\Java\jdk1.8.0_xx\bin`

In "Variable value" field, paste this **IN FRONT** of all the existing directories, followed by a semi-colon (;). **DO NOT DELETE** any existing entries.

```
Variable name  : PATH
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

Now this should resolve.

For Linux Based systems try here.

Note: *The `javac` command invokes the Java compiler.*

The compiler will then generate a bytecode file called `HelloWorld.class` which can be executed in the Java Virtual Machine (JVM). The Java programming language compiler, `javac`, reads source files written in the Java programming language and compiles them into `bytecode` class files. Optionally, the compiler can also process annotations found in source and class files using the `Pluggable` Annotation Processing API. The compiler is a command line tool but can also be invoked using the Java Compiler API.

To run your program, enter `java` followed by the name of the class which contains the `main` method (`HelloWorld` in our example). Note how the `.class` is omitted:

```
$ java HelloWorld
```

Note: *The `java` command runs a Java application.*

This will output to your console:

> Hello, World!

You have successfully coded and built your very first Java program!

Note: In order for Java commands (`java`, `javac`, etc) to be recognized, you will need to make sure:

- A JDK is installed (e.g. Oracle, OpenJDK and other sources)

- Your environment variables are properly set up

You will need to use a compiler (`javac`) and an executor (`java`) provided by your JVM. To find out which versions you have installed, enter `java -version` and `javac -version` on the command line. The version number of your program will be printed in the terminal (e.g. `1.8.0_73`).

A closer look at the Hello World program

The "Hello World" program contains a single file, which consists of a `HelloWorld` class definition, a `main` method, and a statement inside the `main` method.

```
public class HelloWorld {
```

The `class` keyword begins the class definition for a class named `HelloWorld`. Every Java application contains at least one class definition (Further information about classes).

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

This is an entry point method (defined by its name and signature of `public static void main(String[])`) from which the `JVM` can run your program. Every Java program should have one. It is:

- `public`: meaning that the method can be called from anywhere mean from outside the program as well. See Visibility for more information on this.
- `static`: meaning it exists and can be run by itself (at the class level without creating an object).
- `void`: meaning it returns no value. ***Note:*** *This is unlike C and C++ where a return code such as* `int` *is expected (Java's way is* `System.exit()`*).*

This main method accepts:

- An array (typically called `args`) of `String`s passed as arguments to main function (e.g. from command line arguments).

Almost all of this is required for a Java entry point method.

Non-required parts:

- The name `args` is a variable name, so it can be called anything you want, although it is typically called `args`.
- Whether its parameter type is an array (`String[] args`) or Varargs (`String... args`) does not matter because arrays can be passed into varargs.

Note: A single application may have multiple classes containing an entry point (`main`) method. The entry point of the application is determined by the class name passed as an argument to the `java` command.

Inside the main method, we see the following statement:

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

Let's break down this statement element-by-element:

Element	Purpose
`System`	this denotes that the subsequent expression will call upon the `System` class, from the `java.lang` package.

`.`	this is a "dot operator". Dot operators provide you access to a classes members1; i.e. its fields (variables) and its methods. In this case, this dot operator allows you to reference the `out` static field within the `System` class.
`out`	this is the name of the static field of `PrintStream` type within the `System` class containing the standard output functionality.
`.`	this is another dot operator. This dot operator provides access to the `println` method within the `out` variable.
`println`	this is the name of a method within the PrintStream class. This method in particular prints the contents of the parameters into the console and inserts a newline after.
`(`	this parenthesis indicates that a method is being accessed (and not a field) and begins the parameters being passed into the `println` method.
`"Hello, World!"`	this is the String literal that is passed as a parameter, into the `println` method. The double quotation marks on each end delimit the text as a String.
`)`	this parenthesis signifies the closure of the parameters being passed into the `println` method.
`;`	this semicolon marks the end of the statement.

Note: *Each statement in Java must end with a semicolon (`;`).*

The method body and class body are then closed.

```
    }  // end of main function scope
}      // end of class HelloWorld scope
```

Here's another example demonstrating the OO paradigm. Let's model a football team with one (yes, one!) member. There can be more, but we'll discuss that when we get to arrays.

First, let's define our `Team` class:

```
public class Team {
    Member member;
    public Team(Member member) {  // who is in this Team?
        this.member = member;  //  one 'member' is in this Team!
    }
}
```

Now, let's define our `Member` class:

```
class Member {
    private String name;
    private String type;
    private int level; // note the data type here
    private int rank; // note the data type here as well

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }
}
```

Why do we use **`private`** here? Well, if someone wanted to know your name, they should ask you directly, instead of reaching into your pocket and pulling out your Social Security card. This **`private`** does something like that: it prevents outside entities from accessing your variables. You can only return **`private`** members through getter functions (shown below).

After putting it all together, and adding the getters and main method as discussed before, we have:

```
public class Team {
    Member member;
    public Team(Member member) {
        this.member = member;
    }

    // here's our main method
    public static void main(String[] args) {
       Member myMember = new Member("Aurieel", "light", 10, 1);
       Team myTeam = new Team(myMember);
       System.out.println(myTeam.member.getName());
       System.out.println(myTeam.member.getType());
       System.out.println(myTeam.member.getLevel());
       System.out.println(myTeam.member.getRank());
    }
}

class Member {
    private String name;
    private String type;
    private int level;
    private int rank;

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }

    /* let's define our getter functions here */
    public String getName() { // what is your name?
        return this.name; // my name is ...
    }

    public String getType() { // what is your type?
        return this.type; // my type is ...
    }

    public int getLevel() { // what is your level?
        return this.level; // my level is ...
    }

    public int getRank() { // what is your rank?
        return this.rank; // my rank is
    }
}
```

Output:

```
Aurieel
light
10
1
```

Run on ideone

Once again, the `main` method inside the `Test` class is the entry point to our program. Without the `main` method, we cannot tell the Java Virtual Machine (JVM) from where to begin execution of the program.

1 - Because the `HelloWorld` class has little relation to the `System` class, it can only access **`public`** data.