Basics of Java

History of Java

Java is a general-purpose programming language that is class-based, object-oriented, and designed to have as few dependencies as possible. It is intended to let application developers Write Once, Run Anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++ programming languages, but it has fewer low-level facilities than either of them.

Sun Microsystems released the first public implementation as Java 1.0 in 1996. It promised to Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access restrictions. Major web browsers soon incorporated the ability to run Java applets within web pages, and Java quickly became popular. With the advent of Java 2 (released initially as J2SE 1.2 [Java 2 Standart Edition] in December 1998 – 1999), new versions had multiple configurations built for different types of platforms. J2EE (Java 2 Enterprise Edition) included technologies and APIs (Application Programming Interfaces) for enterprise applications typically run in server environments, while J2ME (Java 2 Micro Edition) featured APIs optimized for mobile applications. The desktop version was renamed J2SE. In 2006, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.

As of 2006, Sun released much of its Java Virtual Machine (JVM) as free and open-source software (FOSS), under the terms of the GNU General Public License (GPL). In 2007, Sun finished the process, making all of its JVM's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

Following Oracle Corporation's acquisition of Sun Microsystems in 2009–10, Oracle has described itself as the steward of Java technology with a relentless commitment to fostering a community of participation and transparency. This did not prevent Oracle from filing a lawsuit against Google shortly after that for using Java inside the Android SDK. Java software runs on everything from laptops to data centers, game consoles to scientific supercomputers.

Java Specification

Java language specification

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will not be able to understand it. The Java language specification and the Java API define the Java standards. The application program interface (API), also known as library, contains predefined classes and interfaces for developing Java programs. The Java language specification is a technical definition of the Java programming language's syntax and semantics. You can find the complete Java language specification at Java Language and Virtual Machine Specifications.

What is JVM?

JVM (Java Virtual Machine) is a virtual machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform-independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

What is JRE?

The Java Runtime Environment (JRE) is a software package which bundles the libraries (jars) and the Java Virtual Machine, and other components to run applications written in the Java. JVM is just a part of JRE distributions. To execute any Java application, you need JRE installed in the machine. It's the minimum requirement to execute Java applications on any machine.

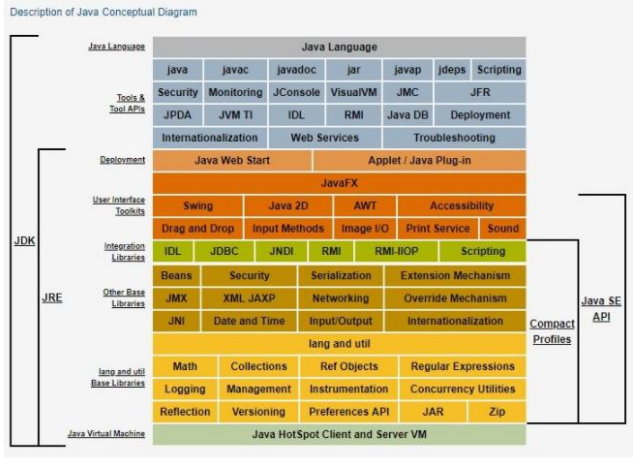
JREs can be downloaded as part of JDKs or you can download them separately. JREs are platform dependent. It means that based on the type of machine (OS and architecture), you will have to select the JRE bundle to import and install.

For example, you cannot install a 64-bit JRE distribution on a 32-bit machine. Similarly, JRE distribution for Windows will not work in Linux; and vice-versa.

What is JDK?

The Java Development Kit (JDK) is a superset of JRE. JDK contains everything that JRE has along with development tools for developing, debugging, and monitoring Java applications. You need JDK when you need to develop Java applications.

Same as JREs, JDKs are also platform dependent. So take care when you download the JDK package for your machine.



A Simple Java Program

A Simple Java Program

Let's begin with a simple Java program that displays the message `Welcome to Java!` on the console.

```
1  public class Welcome {
2      public static void main(String[] args) {
3      // Display message 'Welcome to Java!' on the console
4      System.out.println("Welcome to Java!");
5      }
6  }
7
```

```
1  Welcome to Java!
2
```

Line 1 defines a `class`. Every Java program must have at least one class. Each class has a name. By convention, class names start with an **uppercase letter**. In this example, the class name is `Welcome`.

Line 2 defines the `main` method. The program is executed from the `main` method. A `class` may contain several methods. The `main` method is the entry point where the program begins execution.

A method is a construct that contains statements. The `main` method in this program contains the `System.out.println` statement. This statement displays the string `Welcome to Java!` on the console (line 4). A `string` must be enclosed in **double quotation marks**. Every statement in Java ends with a **semicolon (;)**, known as the statement terminator.

Reserved words, or **keywords**, have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word `class`, it understands that the word after `class` is the name for the `class`. Other reserved words in this program are `public`, `static`, and `void`.

Reserved words, or **keywords**, have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word `class`, it understands that the word after `class` is the name for the `class`. Other reserved words in this program are `public`, `static`, and `void`.

Line 3 is a comment that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements and thus are ignored by the compiler. In Java, comments are preceded by **two slashes (//)** on a line, called a **line comment**, or enclosed between **/* and */** on one or several lines, called a **block comment** or **paragraph comment**. When the compiler sees **//**, it ignores all text after **//** on the same line. When it sees /* , it scans for the next */ and ignores any text between /* and */. Here are examples of comments:

```
// This application program displays Welcome to Java!
/* This application program displays Welcome to Java! */
/* This application program
displays Welcome to Java! */
```

A pair of braces in a program forms a block that groups the program's components. In Java, each block begins with an opening brace ' { ' and ends with a closing brace ' } '. Every `class` has a `class` block that groups the data and methods of the `class`. Similarly, every method has a method block that groups the statements in the method. Blocks can be nested, meaning that one block can be placed within another.

A Simple Java Program

Create, Compile and Run

```java
public class Welcome {
    public static void main(String[] args) {
    // Display message 'Welcome to Java!' on the console
    System.out.println("Welcome to Java!");
    }
}
```

```
Welcome to Java!
```

Steps to create, compile and run your first Java program

Step 1: Open a text editor and write the code as above.

Step 2: Save the file as ***Welcome.java***

Step 3: Open command prompt (Windows)/terminal (Mac/Linux) and go to the directory where you saved your first java program

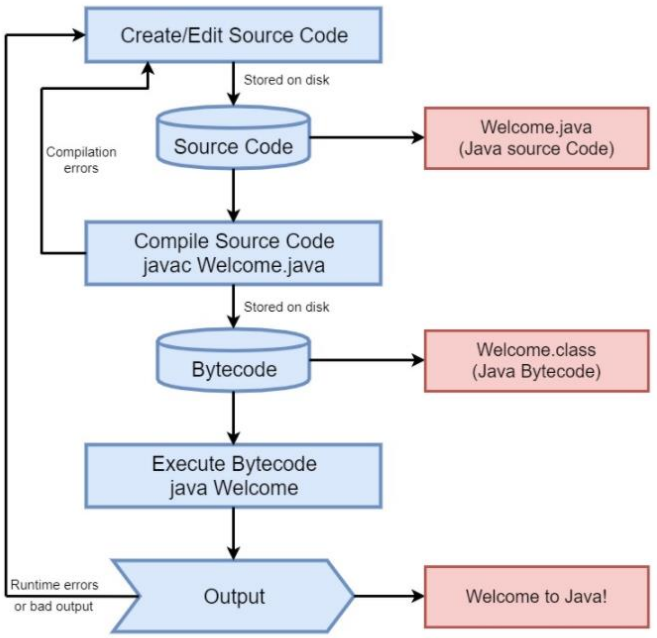
Step 4: Type ***javac Welcome.java*** and press Return(Enter KEY) to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

Step 5: Now type ***java Welcome*** on command prompt/terminal to run your program.

Step 6: You will be able to see *Welcome to Java!* printed on your command prompt/terminal.

Key point: You save a Java program in a **.java** file and compile it into a **.class** file. The **.class** file is executed by the **Java Virtual Machine (JVM).**

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in the figure below. If your program has compilation errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.



Tip: If you execute a class file that does not exist, a **NoClassDefFoundError** will occur. If you execute a class file that does not have a main method or you mistype the main method (e.g., by typing Main instead of main), a **NoSuchMethodError** will occur.

Note: When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the class loader. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called the bytecode verifier to check the validity of the bytecode and to ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure Java class files are not tampered with and do not harm your computer.

Running and Building Locally

What is Building and Compiling?

The term build involves the process of converting source code files into standalone software artifact(s) that can be run on a computer. This is one of the most important steps of a software build which is also named as the compilation process where source code files are converted into executable code.

While for simple programs the process consists of a single file being compiled, for a complex software the source code may consist of many files and may be combined in different ways to produce many different versions.

In the Java world, together with compiling, the "build" process covers all the steps required to create a "deliverable" of your software. This typically includes:

- Generating sources (sometimes).
- Compiling sources.
- Compiling test sources.
- Executing tests (unit tests, integration tests, etc).
- Packaging (into jar, war, ejb-jar, ear).
- Generating reports.

Building JAR Files

Although IDEs like Eclipse or IntelliJ provide support for exporting Java programs as JAR files, we can also do the same thing from the **command line**. JAR stands for `Java Archive` and it is a kind of zip file that holds all contents of a Java application `including class files, resources such as images, sound files, and optional Manifest file`. JAR is a **platform-independent file** including the necessary libraries. JAR files are executable in any operating system like Windows, macOS, or Linux. But this requires a **Java Development Kit**.

Caution: Before going further, be sure to have the Java Development Kit installed. (You can refer to Clarusway's Welcome Kit)

Creating and Running a JAR File From Command Line

First, go to your command-line interface and navigate to the folder that holds your java files. Then run the commands below :

```
MyMac:Desktop home$ cd JavaApp/
MyMac:JavaApp home$ ls
App.java
MyMac:JavaApp home$ javac App.java
MyMac:JavaApp home$ ls
App.class       App.java
MyMac:JavaApp home$ java App
hello world!
MyMac:JavaApp home$ jar -cvfe App.jar App App.class
added manifest
adding: App.class(in = 412) (out= 286)(deflated 30%)
MyMac:JavaApp home$ ls
App.class       App.jar         App.java
MyMac:JavaApp home$ java -jar App.jar
hello world!
MyMac:JavaApp home$
```

In the commands above, with `javac` command, we compile `App.java` and produce `App.class` file. `App.class` file holds the byte array form of our source code. The byte array is a kind of intermediate code between the source code and the machine code. What makes it special is that it makes Java **platform-independent**. JRE consumes it according to the operating system. After the compilation, we run the command `java App`. With this command, we actually run `App.class` file. The extension is automatically added. After that `java App` command runs our application. (We can also do the same with just running `java App.java` on some JDK versions.)

These two commands do not give us a JAR file. To do that we have to run the following commands. If you have already installed Java JDK, you will be able to use jar commands. You can check its presence just by typing `jar` in the command line and hitting enter. If you don't see an error message like `jar command not found` or `jar is not recognized as an internal or external command`, then you have the path to the command.

`jar -cvfe <jar_file_name> <main_class_of_the_application> <java_file>` command creates an executable jar file. If you look at the command options; `e` stands for executable which also requires the main-class part of the command, `f` stands for JAR file name and requires the file name, `v` stands for verbose and `c` stands for create.

The critical part here is to produce an executable jar file. To be able to make a JAR file executable we need to provide the main class of the application with `e` option. This is because the application should know where the entry point is or where to start. Lastly, to make the app run with jar file, we have to run the command `java -jar <jar_file_name>`.

So if you ask what the difference between running the app via the source code or jar file, the jar file way makes the application shippable.