**Core Java Concepts**

**Understanding JVM Architecture**

Understanding JVM architecture and how Java really works under the hood is an important learning for every Java developer in order to effectively make use of the Java ecosystem. This blog post series will provide you with a solid foundation on JVM internals and technologies around the Java ecosystem.

Reference link from Medium article: <https://medium.com/platform-engineer/understanding-jvm-architecture-22c0ddf09722>

**Background**

Designed in 1995 by James Gosling for Sun Microsystems, Java is a multi-paradigm (i.e. object-oriented class-based, structural, imperative, generic, reflective, concurrent) programming language which is loved by millions of developers. On any given ranking index, Java becomes the most popular language for the past 15 years. Tens of thousands of enterprise applications developed in the last 15 years have been mostly written in Java, making it the language of choice for building enterprise-grade production software systems.

Even though I have been using Java since 2015, I recently realized the power of Java ecosystem while doing my final year undergraduate research on Java performance aspects and it motivated to dig deeper into the world of Java. I am planning to write a series of blog posts related to Java internals, performance profiling, server tuning, and many more interesting topics and kindly invite you to stay in touch with this blog. And that’s it for now. Let’s start from primers on Java fundamentals!

**Java Environments**

For almost any programming language, you need a specific environment which comprises of all the necessary components, application programming interfaces, and libraries in order to develop, compile, debug and execute its programs. Java has 2 such environments and everyone working with Java has to start their work after setting up one of these environments on their local development or production environment platforms.

JRE (Java Runtime Environment): the minimum environment needed for running a Java application (no support for developing). It includes JVM (Java Virtual Machine) and deployment tools.

JDK (Java Development Kit): the complete development environment used for developing and executing Java applications. It includes both JRE and development tools.

JRE is meant for users, while JDK is meant for programmers.

**How Java Works**

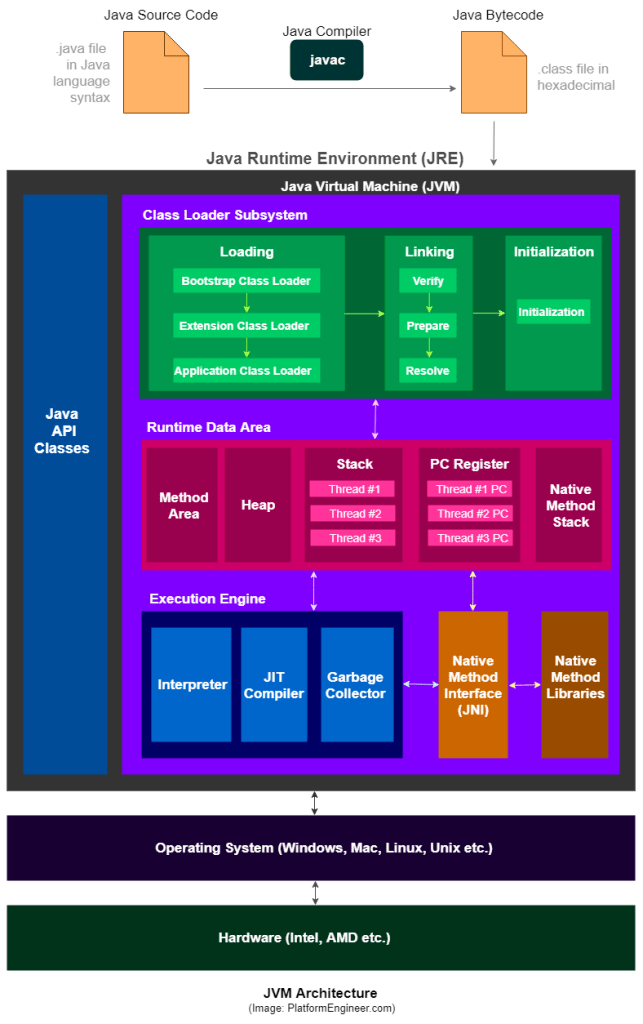
You can start writing a simple Java program with any terminal editor (vim, nano) or GUI editor (gedit, sublime). For a complex Java application, you may need an IDE (Integrated Development Environment) like IntelliJ IDEA, Eclipse, or Netbeans. A typical Java program should contain correct language syntax and .java format. It is recommended to use programming concepts like OOP (Object Oriented Programming) and appropriate architectural patterns for the convenience of structuring and maintaining your Java programs.

The major strength of Java is, it has been designed to run on variety of platforms with the concept WORA — “write once, run anywhere”. Although languages like C++ compile its source code to match only a specific platform and run natively on its OS and hardware, Java source codes are compiled into an intermediate state called bytecode (i.e. a .class file) using the Java Compiler (javac) which comes inbuilt with JDK. This bytecode is in hexadecimal format with opcode-operand lines and JVM can interpret these instructions (without further recompilations) into native machine language which can be understood by the OS and underlying hardware platform. Therefore, bytecode acts as a platform-independent intermediary state which is portable among any JVM regardless of underlying OS and hardware architecture. However, since JVMs are developed to run and communicate with the underlying hardware & OS structure, we need to select the appropriate JVM version for our OS version (Windows, Linux, Mac) and processor architecture (x86, x64).

Most of us know the above story of Java and the problem here is that the most important component of this process — the JVM is taught to us as a black box which can magically interpret bytecode and perform many run-time activities like JIT (Just-in-time) compilation & GC (Garbage Collection) during the program execution. In the next sections, let’s reveal how JVM works.

**JVM Architecture**

JVM is only a specification, and its implementation is different from vendor to vendor. For now, let’s understand the commonly-accepted architecture of JVM as defined in the specification.



**1) Class Loader Subsystem**

The JVM resides on the RAM. During execution, using the Class Loader subsystem, the class files are brought on to the RAM. This is called Java’s dynamic class loading functionality. It loads, links, and initializes the class file (.class) when it refers to a class for the first time at runtime (not compile time).

1.1) **Loading**

Loading compiled classes (.class files) into memory is the major task of Class Loader. Usually, the class loading process starts from loading the main class (i.e. class with static main() method declaration). All the subsequent class loading attempts are done according to the class references in the already-running classes as mentioned in the following cases:

• When bytecode make a static reference to a class (e.g. System.out)

• When bytecode create a class object (e.g. Person person = new Person("John"))

There are 3 types of class loaders (connected with inheritance property) and they follow 4 major principles.

1.1.1) **Visibility Principle**

This principle states that Child Class Loader can see the class loaded by Parent Class Loader, but a Parent Class Loader cannot find the class loaded by Child Class Loader.

1.1.2) **Uniqueness Principle**

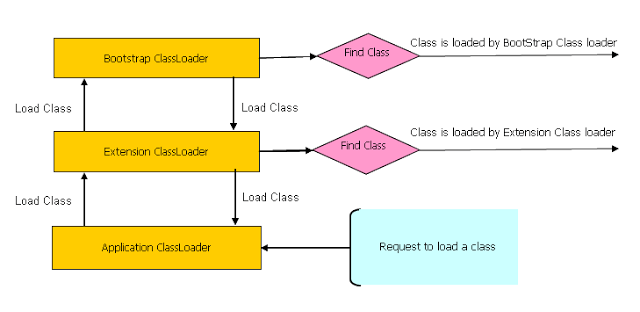
This principle states that a class loaded by parent should not be loaded by Child Class Loader again and ensure that duplicate class loading does not occur.

1.1.3) **Delegation Hierarchy Principle**

In order to satisfy above 2 principles, JVM follows a hierarchy of delegation to choose the class loader for each class loading request. Here, starting from the lowest child level, Application Class Loader delegates the received class loading request to Extension Class Loader and then Extension Class Loader delegates the request to Bootstrap Class Loader. If the requested class found in Bootstrap path, the class is loaded. Otherwise the request again transfers back to Extension Class Loader level to find the class from Extension path or custom-specified path. If it also fails, the request comes back to Application Class Loader to find the class from System class path and if Application Class Loader also fails to load the requested class, then we get the run time exception — java.lang.ClassNotFoundException .

1.1.4) **No Unloading Principle**

Even though a Class Loader can load a class, it cannot unload a loaded class. Instead of unloading, the current class loader can be deleted, and a new class loader can be created



**Java Class Loaders — Delegation Hierarchy Principle (Image: StackOverflow.com)**

**Bootstrap Class Loader** loads standard JDK classes from rt.jar such as core Java API classes present in the bootstrap path — $JAVA\_HOME/jre/lib directory (e.g. java.lang.\* package classes). It is implemented in native languages like C/C++ and acts as parent of all class loaders in Java.

**Extension Class Loader delegates** class loading request to its parent, Bootstrap and if unsuccessful, loads classes from the extensions directories (e.g. security extension functions) in extension path — $JAVA\_HOME/jre/lib/ext or any other directory specified by the **java.ext.dirs** system property. This Class Loader is implemented in Java by the sun.misc.Launcher$ExtClassLoader class.

**System/Application Class Loader** loads application specific classes from system class path, that can be set while invoking a program using -cp or -classpath command line options. It internally uses Environment Variable which mapped to **java.class.path**. This Class Loader is implemented in Java by the **sun.misc.Launcher$AppClassLoader class.**

**NOTE**:

Apart from the 3 major Class Loaders discussed above, a programmer can directly create a User-defined Class Loader on the code itself.

This guarantees the independence of applications through class loader delegation model.

This approach is used in web application servers like Tomcat to make web apps and enterprise solutions **run independently**.

* Each Class Loader has **its namespace that stores the loaded classes**.
* When a Class Loader loads a **class**, it searches the class based on **FQCN** (Fully Qualified Class Name) stored in the namespace to check whether or not the class has been already loaded. Even if the class has an identical **FQCN** but a different namespace, it is regarded as a different class.
* A different namespace means that the class has been loaded by another Class Loader.

**Examples:**

Reference link : <https://incusdata.com/blog/java-class-loaders-part-1>

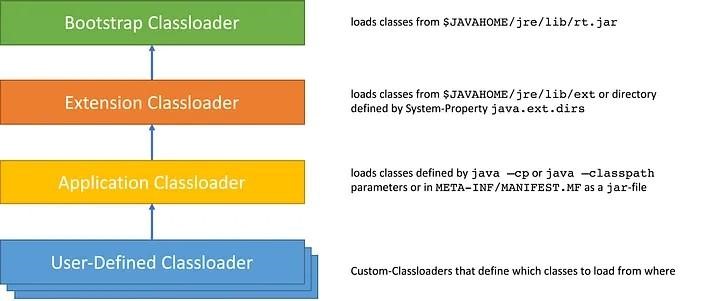
|  |
| --- |
| import java.sql.DriverManager;  import java.util.ArrayList;  public class PrintClassLoaders {  public static void main(String args[]) {  System.out.println("java.ext.dirs: "  + System.getProperty("java.ext.dirs"));  System.out.println("Classloader of ArrayList: "  + ArrayList.class.getClassLoader());  System.out.println("Classloader of String: "  + String.class.getClassLoader());  System.out.println("Classloader of this class: "  + PrintClassLoaders.class.getClassLoader());  System.out.println("Classloader of DriverManager: "  + DriverManager.class.getClassLoader());  System.out.println("Classloader of com.sun.nio.zipfs.ZipInfo: "  + com.sun.nio.zipfs.ZipInfo.class.getClassLoader());  }  }  **Java 8 Output:**  Running this in Java 8 will give an output similar to the following (object addresses will differ on your machine):  java.ext.dirs: d:\jdk8\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext  Classloader of String: null  Classloader of ArrayList: null  Classloader of this class: sun.misc.Launcher$AppClassLoader@73d16e93  Classloader of DriverManager: null  Classloader of com.sun.nio.zipfs.ZipInfo: [sun.misc.Launcher$ExtClassLoader@70dea4e](mailto:sun.misc.Launcher$ExtClassLoader@70dea4e)  **Java 11 Output**  Running the same code in Java 11 without recompiling will give an output similar to the following:  java.ext.dirs: null  Classloader of ArrayList: null  Classloader of String: null  Classloader of this class: jdk.internal.loader.ClassLoaders$AppClassLoader@30946e09  Classloader of DriverManager: jdk.internal.loader.ClassLoaders$PlatformClassLoader@5cbc508c  Exception in thread "main" java.lang.NoClassDefFoundError: com/sun/nio/zipfs/ZipInfo  at PrintClassLoaders.main(PrintClassLoaders.java:24)  Caused by: java.lang.ClassNotFoundException: com.sun.nio.zipfs.ZipInfo  ... |

**Important Note:**

* We can see that the class loaders **for String, ArrayList and DriverManager all return null**. This generally represents the bootstrap class loader.
* The PrintClassLoaders application class returns **sun.misc.Launcher$AppClassLoader.** This is the application/system class loader.
* The ZipInfo class loader returns sun.misc.Launcher$ExtClassLoader. This is the extension class loader. The ZipInfo class was chosen at random to illustrate this.

|  |
| --- |
| The PrintClassLoaders application class is loaded by the application/system class loader. This class loader is different to Java 8. It is now jdk.internal.loader.ClassLoaders$AppClassLoader. The **AppClassLoader** is responsible for loading classes from the application classpath and module path. |

**Creating the custom class loader and loading our own class with custom class loader:**



* For illustration purposes, let’s say we need to load classes from a file using a custom class loader. We need to extend the ClassLoader class and override the findClass() method:
* The default implementation of the method searches for classes in the following order:

1. Invokes the findLoadedClass(String) method to see if the class is already loaded.
2. Invokes the loadClass(String) method on the parent class loader.
3. Invoke the findClass(String) method to find the class.

**Example:**

|  |
| --- |
| package com.demo.satiee;  import java.io.ByteArrayOutputStream;  import java.io.File;  import java.io.IOException;  import java.io.InputStream;  public class CustomClassLoader extends ClassLoader {  @Override  public Class findClass(String name) throws ClassFormatError {  byte[] b = loadClassFromFile(name);  return defineClass(name, b, 0, b.length);  }  private byte[] loadClassFromFile(String fileName) {  InputStream inputStream = getClass().getClassLoader()  .getResourceAsStream(fileName.replace('.', '/') + ".class");  byte[] buffer;  ByteArrayOutputStream byteStream = new ByteArrayOutputStream();  int nextValue = 0;  //com/demo/satiee/Movie.class  System.out.println(fileName.replace('.', File.separatorChar) + ".class");  try {  while ((nextValue = inputStream.read()) != -1) {  byteStream.write(nextValue);  }  } catch (IOException e) {  e.printStackTrace();  }  buffer = byteStream.toByteArray();  return buffer;  }  } |

|  |
| --- |
| package com.demo.satiee;  import java.lang.reflect.Method;  public class ExampleClassLoaderMain {  public static void main(String[] args) {  CustomClassLoader customClassLoader = new CustomClassLoader();  try {  Object obj;  obj = customClassLoader.findClass("com.demo.satiee.Movie").newInstance();  System.out.println(obj.hashCode());  Method[] methods = obj.getClass().getDeclaredMethods();  System.out.println(String.format("Methods of %s class:", obj.getClass().getName()));  for (Method method : methods) {  System.out.println(method.getName());  }  } catch (ClassFormatError e) {  e.printStackTrace();  } catch (IllegalAccessException e) {  e.printStackTrace();  } catch (InstantiationException e) {  e.printStackTrace();  }  }  } |

Q) How to break single ton class behavior using custom class loader in java?

* On the other hand, there’s another way to break the singleton pattern, which cannot be solved using either of above, and any way that I could think of. That is to use multiple class loaders.
* When the same class is loaded by two different class loaders, that same class is treated as if they are two different classes.
* That is because the Java identifies unique classes not only using it’s fully qualified name, but also with the class loader which loaded the class. If our singleton above is loaded by two class loaders, there will be two instances of it.
* That being said, use of Singletons should be done with care, especially when the singleton maintains state. In distributed environments such as clusters (each VM will have its own singleton instance), relying on the “singleton-ness”of singletons could lead to hard to find bugs.
* Useful reference link to Prevent Breaking a Singleton Class Pattern- <https://dzone.com/articles/prevent-breaking-a-singleton-class-pattern>

**Examples: But it’s not working as of now**

|  |
| --- |
| CustomClassLoader customClassLoader = new CustomClassLoader();  CustomClassLoaderSingleton customClassLoaderSingleton = new CustomClassLoaderSingleton();  try {  Object obj1;  Object obj2;  Class classSingleton1 = customClassLoader.findClass("com.demo.satiee.Singleton");  Constructor constructor1 = classSingleton1.getDeclaredConstructor();  constructor1.setAccessible(true);  obj1 = classSingleton1.newInstance();      Class classSingleton2 = customClassLoaderSingleton.findClass("com.demo.satiee.Singleton");  Constructor constructor2 = classSingleton2.getDeclaredConstructor();    constructor2.setAccessible(true);    obj2 = classSingleton2.newInstance();    System.out.println("obj1.hashCode()" + obj1.hashCode());  System.out.println("obj12.hashCode()" + obj2.hashCode());  Method[] methods = obj1.getClass().getDeclaredMethods(); |

Q) What is class path in java?

* To simplify, the Java Classpath is just a collection of paths (directories and JAR files) used by the Java Compiler to compile and by Java Virtual Machine (JVM) to look for classes or other resources that **are required by a Java program at runtime.**
* Our compiled classes can be located inside a directory or a JAR file. And accordingly, to **compile/run** the java app we need to provide them in the **classpath**

**PATH vs CLASSPATH**

The PATH is used by **the operating system** to find executable files.

The CLASSPATH is used by the **JVM** to find **class files needed by a Java program.**

**Setting the Classpath**

How to set up the Classpath

There are **3 ways yo**u set up the Classpath in Java:

1. using command-line arguments
2. using environment variables
3. using IDEs

**Examples:**

Let’s take a look at how to compile and run the following file:

|  |
| --- |
| package edu.example;  public class Main {  public static void main(String[] args) {  System.out.println("Hello");  }  } |

**Using the “-cp” or “-classpath” options**

To run a Java application using the “-cp” or “-classpath” options, you need to specify the classpath when you invoke the “java” command

**Commands to run java program:**

|  |
| --- |
| javac src/main/java/\*\*/\*.java -d target/classes --------------------- Compilation  java -cp ./target/classes edu.example.Main  # or  java -classpath ./target/classes edu.example.Main |

**Using the “CLASSPATH” environment variable**

* To run a Java application using the “CLASSPATH” environment variable, you need to set up “CLASSPATH” environment variable before you invoke the “java” command

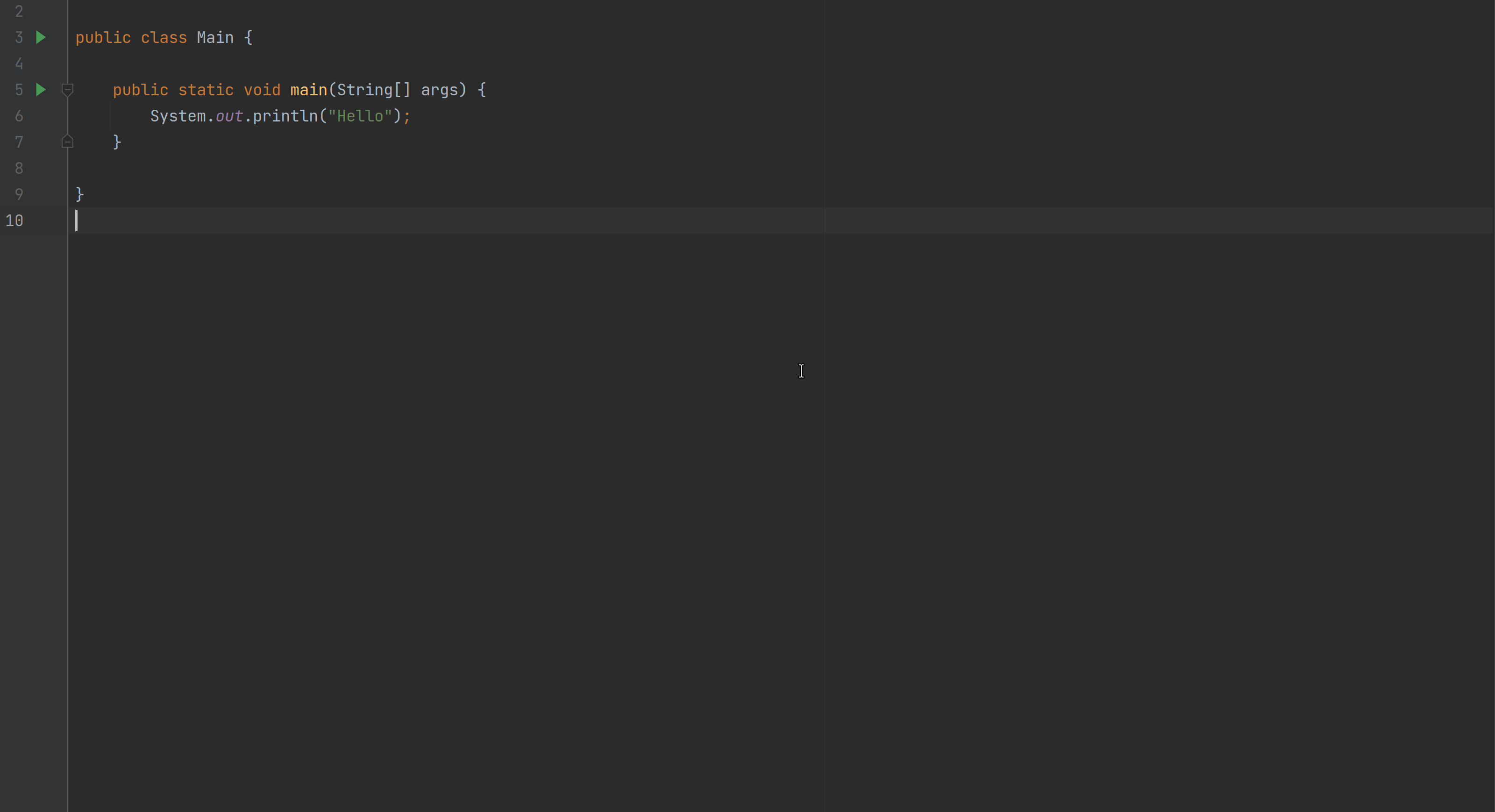
**Please note:** There are others options you could use for “compiling multiple source files in the same directory”[1]

|  |
| --- |
| javac src/main/java/\*\*/\*.java -d target/classes  export CLASSPATH=./target/classes  java edu.example.Main |

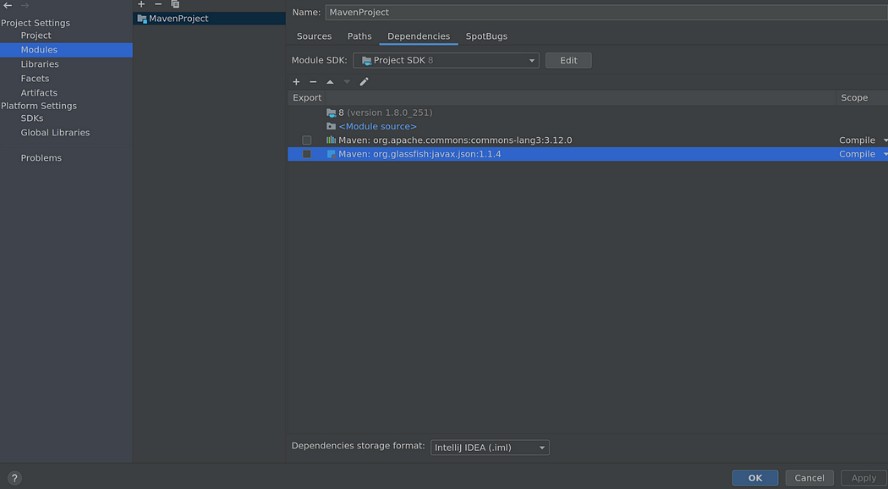
Please note: “The preferred way to specify the class path is by using the -cp command line switch”[2]

**Using IntelliJ IDEA**

If you use IntelliJ IDEA[3] — no need to think much about it because it’s quite easy.



You may modify the classpath in the “Project Structure” settings.



**What if our code uses an external library?**

* Imagine you have added some dependencies (class or method from an external library) in your code.
* Then to be able to compile you definitely need that class available in the classpath during compilation and during the runtime. In this case, we will provide the path to JARs (with class and method dependencies) and to our compiled class.

|  |
| --- |
| +import static org.apache.commons.lang3.StringUtils.capitalize;  +  public class Main {  public static void main(String[] args) {  - System.out.println("Hello");  + System.out.println(Json.createObjectBuilder()  + .add("text", "Hello, " +  + capitalize(args.length > 0 ? args[0] : "anonymous"))  + .build());  }  } |

|  |
| --- |
| javac -cp "lib/commons-lang3-3.12.0.jar:lib/javax.json-1.1.4.jar" src/main/java/\*\*/\*.java -d target/classes  java -cp "lib/commons-lang3-3.12.0.jar:lib/javax.json-1.1.4.jar:target/classes" ed |

**Please note:** Sometimes you don’t use the same dependencies for compilation and in runtime (for e.g. jar with API interfaces — for compilation, jar with implementation classes — for runtime). And it’s also a good thing to know “What is the difference between compile time and run time dependencies in Java”[4]

**Conclusion**

1. In conclusion, the Java classpath is an essential component of Java development. It specifies where the Java Virtual Machine should look for class files and other resources needed by a Java program.
2. We have seen that the classpath can be set in various ways, including using environment variables and command-line options.
3. A misconfigured classpath can result in errors and make it difficult to run Java programs successfully. By understanding the classpath and how to configure it correctly, developers can avoid these issues and ensure that their Java programs run smoothly and reliably.
4. Understanding how the classpath works and how to configure it correctly can save developers a lot of time and frustration.
5. Reference Link: <https://medium.com/@rostyslav.ivankiv/what-is-java-classpath-what-every-developer-should-know-e5f648bde862>

Q) When do we use /set the class path in java? Is it during javac command or java command running?

**Important Note:**

* We can use the class path in both compile time and run time places. Compile time class path will be useful to compiler to check where our own or third party classes available.
* Run time class path is use full to application/system class loader or JVM to load the .class files dynamically at run time.

1.2) Linking

Linking involves in verifying and preparing a loaded class or interface, its direct superclasses and superinterfaces, and its element type as necessary, while following the below properties.

* A class or interface must be completely loaded before it is linked.
* A class or interface must be completely verified and prepared before it initialized (in the next step).
* If an error occurs during linking, it is thrown at a point in the program where some action will be taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error.

Linking occurs in 3 stages as below.

1. **Verification**:

* ensure the correctness of .class file (is the code properly written according to Java Language Specification? is it generated by a valid compiler according to JVM specifications?).
* This is the most complicated test process of the class load processes, and takes the longest time.
* Even though linking slows down the class loading process, it avoids the need to perform these checks for multiple times when executing bytecode, hence makes the overall execution efficient and effective. If verification fails, it throws runtime errors (java.lang.VerifyError). For instance, the following checks are performed.

|  |
| --- |
| - consistent and correctly formatted symbol table  - final methods / classes not overridden  - methods respect access control keywords  - methods have correct number and type of parameters  - bytecode doesn’t manipulate stack incorrectly  - variables are initialized before being read  - variables are a value of the correct type |

1. **Preparation:**

* Allocate memory for static storage and any data structures used by the JVM such as method tables. Static fields are created and initialized to their default values, however, no initializers or code is executed at this stage as that happens as part of initialization.

1. **Resolution:**

* Replace symbolic references from the type with direct references.
* It is done by searching into method area to locate the referenced entity.

**1.3) Initialization**

* Here, the initialization logic of each loaded class or interface will be executed (e.g. calling the constructor of a class).
* Since **JVM is multi-threaded**, initialization of a class or interface should happen very carefully with proper synchronization to avoid some other thread from trying to initialize the same class or interface at the same time (i.e. make it thread safe).
* This is the **final phase** of class loading where all the static variables are assigned with their original values defined in the code and the static block will be executed (if any).
* This is executed line by line from top to bottom in a class and from parent to child in class hierarchy.

**2) Runtime Data Area:**

* Runtime Data Areas are the memory areas assigned when the JVM program runs on the OS.
* In addition to reading .class files, the **Class Loader subsystem generates corresponding binary data and save the following information** in the Method area for each class separately.
* Fully qualified name of the loaded class and its immediate parent class
* Whether .class file is related to a Class/Interface/Enum
* Modifiers, static variables, and method information etc

Then, for every loaded .class file, **it creates exactly one object of Class** to represent the file in the Heap memory as defined **in java.lang package.**

This **Class object can be used to read class level information** (class name, parent name, methods, variable information, static variables etc.) later in our code.

**2.1) Method Area (Shared among Threads)**

* This is a shared resource (only 1 method area per JVM).
* All JVM threads share this same Method area, so the access to the Method data and the process of dynamic linking must be thread safe.
* Method area stores class level data (including static variables) such as:

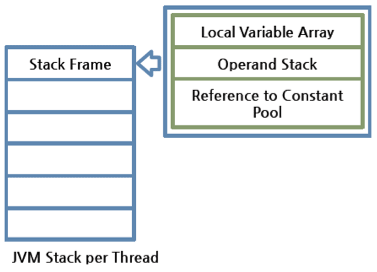
1. **Classloader reference**
2. **Run time constant pool** — Numeric constants, field references, method references, attributes; As well as the constants of each class and interface, it contains all references for methods and fields. When a method or field is referred to, the JVM searches the actual address of the method or field on the memory by using the runtime constant pool.
3. **Field data** — Per field: name, type, modifiers, attributes
4. **Method data** — Per method: name, return type, parameter types (in order), modifiers, attributes
5. **Method code** — Per method: bytecodes, operand stack size, local variable size, local variable table, exception table; Per exception handler in exception table: start point, end point, PC offset for handler code, constant pool index for exception class being caught

**2.2) Heap Area (Shared among Threads)**

* This is also a shared resource (only 1 heap area per JVM).
* Information of all objects and their corresponding instance variables and arrays are stored in the Heap area.
* Since the Method and Heap areas share memory for multiple threads, the data stored in Method & Heap areas are not thread safe.
* Heap area is a great target for GC.

**2.3) Stack Area (per Thread)**

* This is **not** a shared resource.
* For every JVM thread, when the thread starts, a separate runtime stack gets created in order to store method calls.
* For every such method call, one entry will be created and added (pushed) into the top of **runtime stack and such entryit is called a Stack Frame**.
* Each stack frame has the reference for local variable array, Operand stack, and runtime constant pool of a class where the method being executed belongs.
* The size of local variable array and Operand stack is determined while compiling.
* Therefore, the size of stack frame is fixed according to the method.
* The frame is removed (popped) when the method returns normally or if an uncaught exception is thrown during the method invocation.
* Also note that if any exception occurs, each line of the stack trace (**shown as a method such as printStackTrace**()) expresses one stack frame. The Stack area is thread safe since it is not a shared resource.



A Stack Frame is divided into three sub-entities:

**Local Variable Array —**

* It has an index starting from 0. For a particular method, how many local variables are involved and the corresponding values are stored here.
* 0 is the reference of a class instance where the method belongs. From 1, the parameters sent to the method are saved. After the method parameters, the local variables of the method are saved.

**Operand Stack —**

* This acts as a runtime workspace to perform any intermediate operation if there’s a requirement.
* Each method exchanges data between the Operand stack and the local variable array, and pushes or pops other method invoke results.
* The necessary size of the Operand stack space can be determined during compiling.
* Therefore, the size of the Operand stack can also be determined during compiling.

**Frame Data —**

* All symbols related to the method are stored here.
* For exceptions, the catch block information will also be maintained in the frame data.

**Important Point:** Difference between **StackOverflowError and OutOfMemeroyError** ?

Since these are runtime stack frames, after a thread terminates, its stack frame will also be destroyed by JVM.

* A stack can be a dynamic or fixed size.
* If a thread requires a larger stack than allowed a **StackOverflowError** is thrown.
* If a thread requires a new frame and there isn’t enough memory to allocate it then an **OutOfMemoryError** is thrown.

2.4) **PC Registers (per Thread)**

* For each JVM thread, when the thread starts, a separate PC (Program Counter) Register gets created in order to hold the address of currently-executing instruction (memory address in the Method area).
* If the current method is native then the PC is undefined.
* Once the execution finishes, the PC register gets updated with the address of next instruction.

2.5) **Native Method Stack (per Thread)**

* There is a direct mapping between a **Java thread and a native operating system thread**.
* After preparing all the state for a Java thread, a separate native stack also gets created in order to store native method information (often written in C/C++) invoked through JNI (Java Native Interface).
* Once the native thread has been created and initialized, it invokes the run()method in the Java thread.
* When the run() method returns, uncaught exceptions (if any) are handled, then the native thread confirms whether the JVM needs to be terminated as a result of the thread terminating (i.e. is it the last non-deamon thread).
* When the thread terminates, all resources for both the native and Java threads are released.
* The native thread is reclaimed once the Java thread terminates.
* The operating system is therefore responsible for scheduling all threads and dispatching them to any available CPU.

**3) Execution Engine**

The actual execution of the bytecode occurs here.

Execution Engine executes the instructions in the bytecode line-by-line by reading the data assigned to above runtime data areas.

3.1) Interpreterx

* The interpreter interprets the bytecode and executes the instructions one-by-one.
* Hence, it can interpret one bytecode line quickly, but executing the interpreted result is a slower task.
* The disadvantage is that when one method is called multiple times, each time a new interpretation and a slower execution are required.

3.2) Just-In-Time (JIT) Compiler

* If only the interpreter is available, when one method **is called multiple times**, each time the interpretation will also occur, which is a redundant operation if handled efficiently.
* This has become possible **with JIT compiler**.
* First, it compiles the **entire bytecode to native code** (machine code).
* Then for repeated method calls, it directly provides the native code and the execution using native code is much faster than interpreting instructions one by one.
* The native code is stored in **the cache**, thus the compiled code can be executed quicker.

Q) Is Java is pass by value or pass by reference?

<https://www.baeldung.com/java-pass-by-value-or-pass-by-reference>

**1.Introduction:**

* The two most prevalent modes of passing arguments to methods are “passing-by-value” and “passing-by-reference”.
* Different programming languages use these concepts in different ways. As far as Java is concerned, everything is strictly Pass-by-Value.

2.**Pass-by-Value vs Pass-by-Reference:**

Let’s start with some of the different mechanisms for passing parameters to functions:

* value
* reference
* result
* value-result
* name

The two most common mechanisms in modern programming languages are “Pass-by-Value” and “Pass-by-Reference”. Before we proceed, let’s discuss these first:

**2.1. Pass-by-Value:**

When a parameter is pass-by-value, the **caller** and the **callee** method operate on two different variables which are **copies of each other.**

* 1. Any changes to one variable don’t modify the other.
  2. It means that while calling a method, parameters passed to the callee method will be clones of original parameters.
  3. Any modification done in callee method will have no effect on the original parameters in caller method.

**2.2. Pass-by-Reference:**

When a parameter is **pass-by-reference**, the **caller and the callee** operate on the same object.

1. It means that when a variable is pass-by-reference, the unique identifier of the object is sent to the method.
2. Any changes to the parameter’s instance members will result in that change being made to the original value.

**3.Parameter Passing in Java:**

The fundamental concepts in any programming language are **“values” and “references”.**

* In Java, Primitive variables **store the actual values**, whereas Non-Primitives store the **reference variables** which point to the addresses of the **objects** they’re referring to.
* Both values and references are stored in the **stack memory.**

**Important points:**

Arguments in Java are always passed-by-value.

1. During method invocation, a copy of each argument, whether its a value or reference, is created in stack memory which is then passed to the method.
2. In case of primitives, the **value is simply copied inside stack memory** which is then passed to the **callee** method; in case of non-primitives, a reference in **stack memory points** to the actual data which resides in the heap.
3. When we pass an object, the **reference in stack memory is copied and the new reference is passed to the method**.

Let’s now see this in action with the help of some code examples.

**3.1 Passing Primitive Types:**

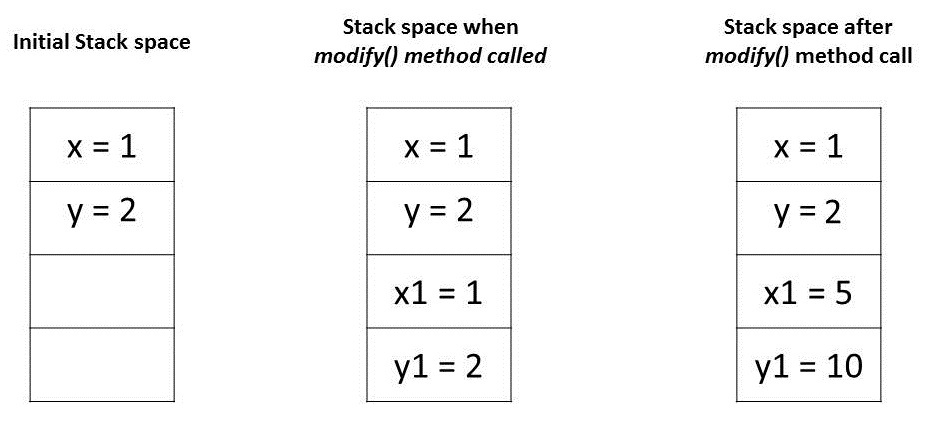
1. The Java Programming Language features **eight primitive data** types.
2. Primitive variables are directly **stored in stack memory.**
3. Whenever any variable of primitive data type is passed as an argument, the actual parameters are **copied to formal arguments** and these **formal arguments accumulate** their own space in stack memory.
4. The lifespan of these formal parameters lasts only as long as that method is running, and upon returning, these formal arguments are cleared away from the stack and are discarded.

Let’s try to understand it with the help of a code example:

|  |
| --- |
| public class PrimitivesUnitTest {  @Test  public void whenModifyingPrimitives\_thenOriginalValuesNotModified() {    int x = 1;  int y = 2;    // Before Modification  assertEquals(x, 1);  assertEquals(y, 2);    modify(x, y);    // After Modification  assertEquals(x, 1);  assertEquals(y, 2);  }    public static void modify(int x1, int y1) {  x1 = 5;  y1 = 10;  }  } |

**Explanation:**

* Let’s try to understand the assertions in the above program by analyzing how these values are stored in memory:
* The variables “x” and “y” in the main method are primitive types and their values are directly stored in the stack memory.
* When we call method **modify(),** an exact copy for each of these variables is **created and stored at a different location in** the stack memory.
* Any modification to these **copies affects only them and** leaves the original variables unaltered.



**3.2. Passing Object References:**

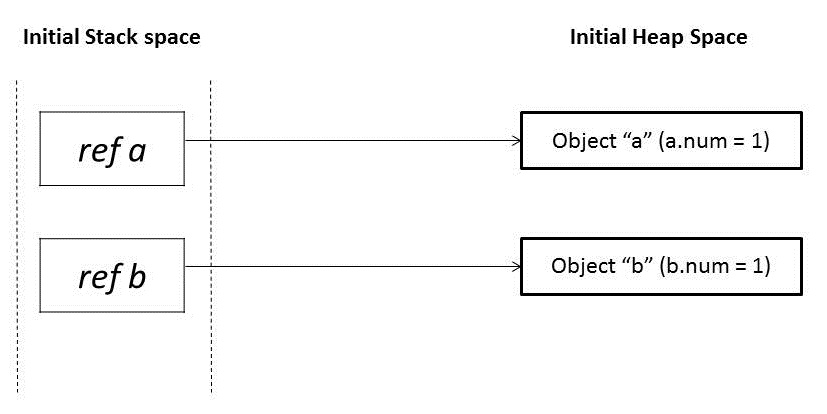
1. In Java, all objects are dynamically stored in Heap space under the hood.
2. These objects are referred from references called reference variables.
3. A Java object, in contrast to Primitives, is stored in two stages.
4. The reference variables are stored in stack memory and the object that they’re referring to, are stored in a Heap memory.
5. Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.
6. As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object.
7. However, if we allocate a new object to the passed reference variable, then it won’t be reflected in the original object.

Let’s try to comprehend this with the help of a code example:

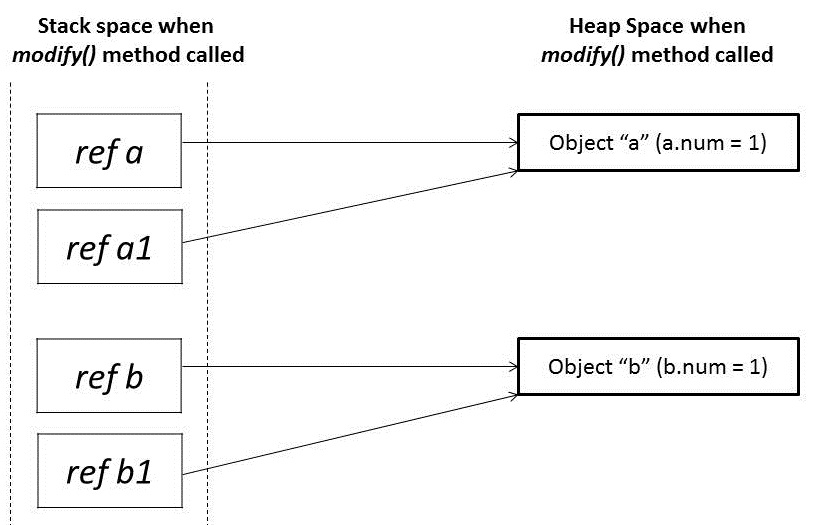
|  |
| --- |
| public class NonPrimitivesUnitTest {    @Test  public void whenModifyingObjects\_thenOriginalObjectChanged() {  Foo a = new Foo(1);  Foo b = new Foo(1);  // Before Modification  assertEquals(a.num, 1);  assertEquals(b.num, 1);    modify(a, b);    // After Modification  assertEquals(a.num, 2);  assertEquals(b.num, 1);  }    public static void modify(Foo a1, Foo b1) {  a1.num++;    b1 = new Foo(1);  b1.num++;  }  }    class Foo {  public int num;    public Foo(int num) {  this.num = num;  }  } |

**Explanation:**

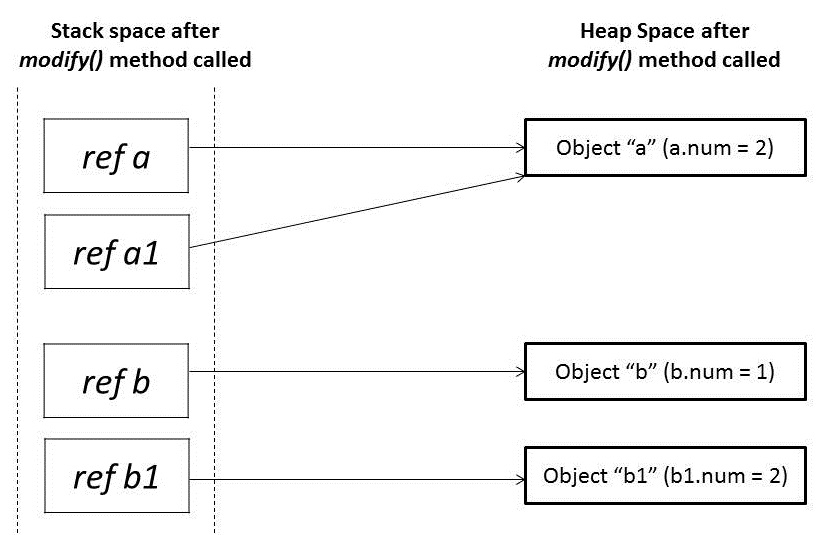
1. Let’s analyze the assertions in the above program.
2. We have passed objects a and b in modify() method that has the same value 1.
3. Initially, these object references are pointing to two distinct object locations in a heap space:



When these references a and b are passed in the modify() method, it creates mirror copies of those references a1 and b1 which point to the same old objects:



* In the modify() method, when we modify reference a1, it changes the original object.
* However, for a reference b1, we have assigned a new object.
* So it’s now pointing to a new object in heap memory.
* Any change made to b1 will not reflect anything in the original object:



We learned that parameter passing in Java is always Pass-by-Value. However, the context changes depending upon whether we’re dealing with Primitives or Objects:

1. For Primitive types, parameters are pass-by-value
2. For Object types, the object reference is pass-by-value

Q) How a variable, class, method will be stored inside JVM memory allocations? Java Memory Management? Garbage collectors? What is difference between perm gem and meta space ?

<https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973>

Q) How to create our own immutable class with List/Set/Map of properties inside that class?

<https://salithachathuranga94.medium.com/implement-immutable-classes-with-java-df5b5b66ffd9>

Q) Threads, Multi-threading, Thread pool, Thread schedulers,Executor Framework:

how thread will get created in executor service framework java internally - <https://medium.com/codex/executorservice-internal-working-in-java-7b286882f54e>

Q) How to allow prototype behavior into Singleton class?

Q) Java Inheritance?

Q) why “String” is immutable?

Q) What is cloning? difference between Shallow cloning and Deep cloning?

Q) Why Generic and how to use?