**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**

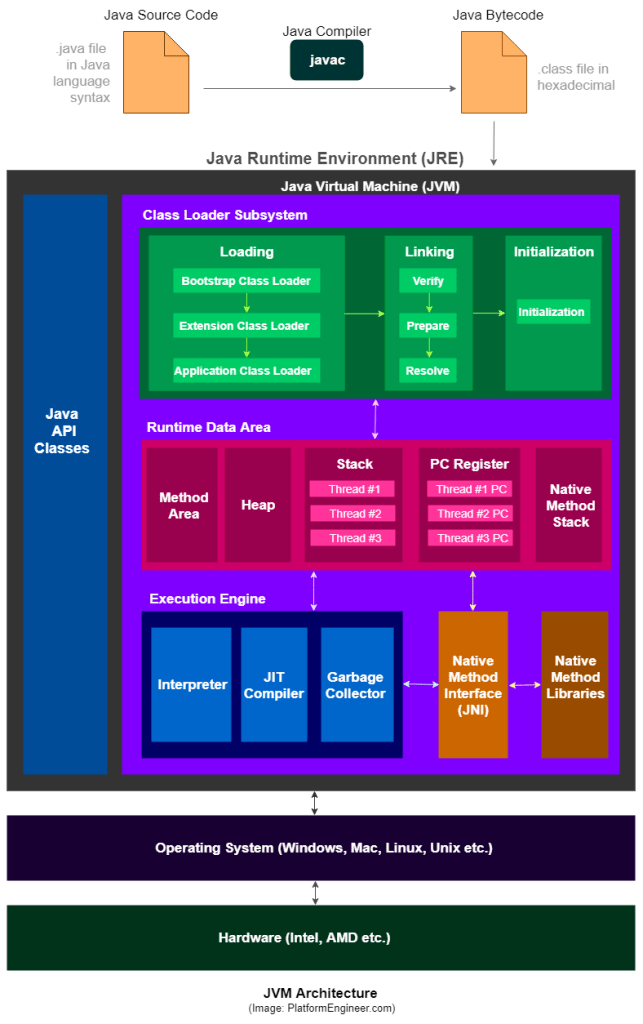
You can start writing a simple Java program with any terminal editor (vim, nano) or GUI editor (gedit, sublime). For a complex Java application, you may need an IDE (Integrated Development Environment) like IntelliJ IDEA, Eclipse, or Netbeans. A typical Java program should contain correct language syntax 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**

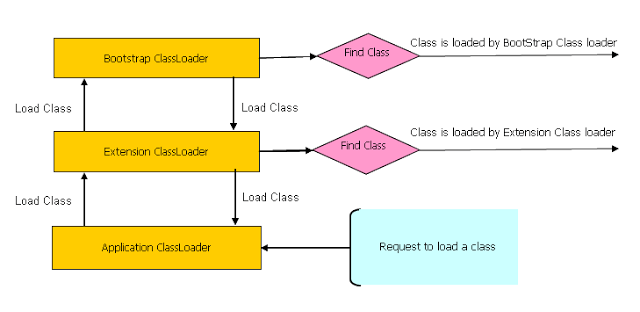
This principle states that a class loaded by parent should not be loaded by Child Class Loader again and ensure that duplicate class loading does not occur.

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>

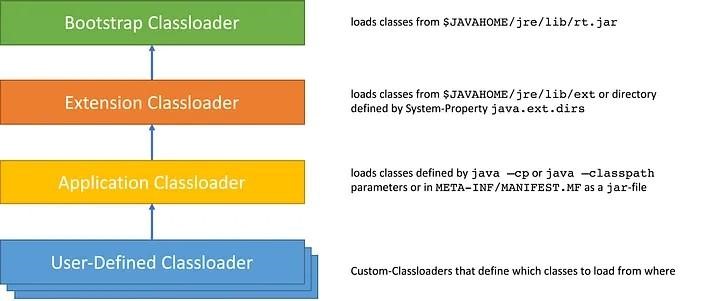
|  |
| --- |
| import java.sql.DriverManager;  import java.util.ArrayList;  public class PrintClassLoaders {  public static void main(String args[]) {  System.out.println("java.ext.dirs: "  + System.getProperty("java.ext.dirs"));  System.out.println("Classloader of ArrayList: "  + ArrayList.class.getClassLoader());  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

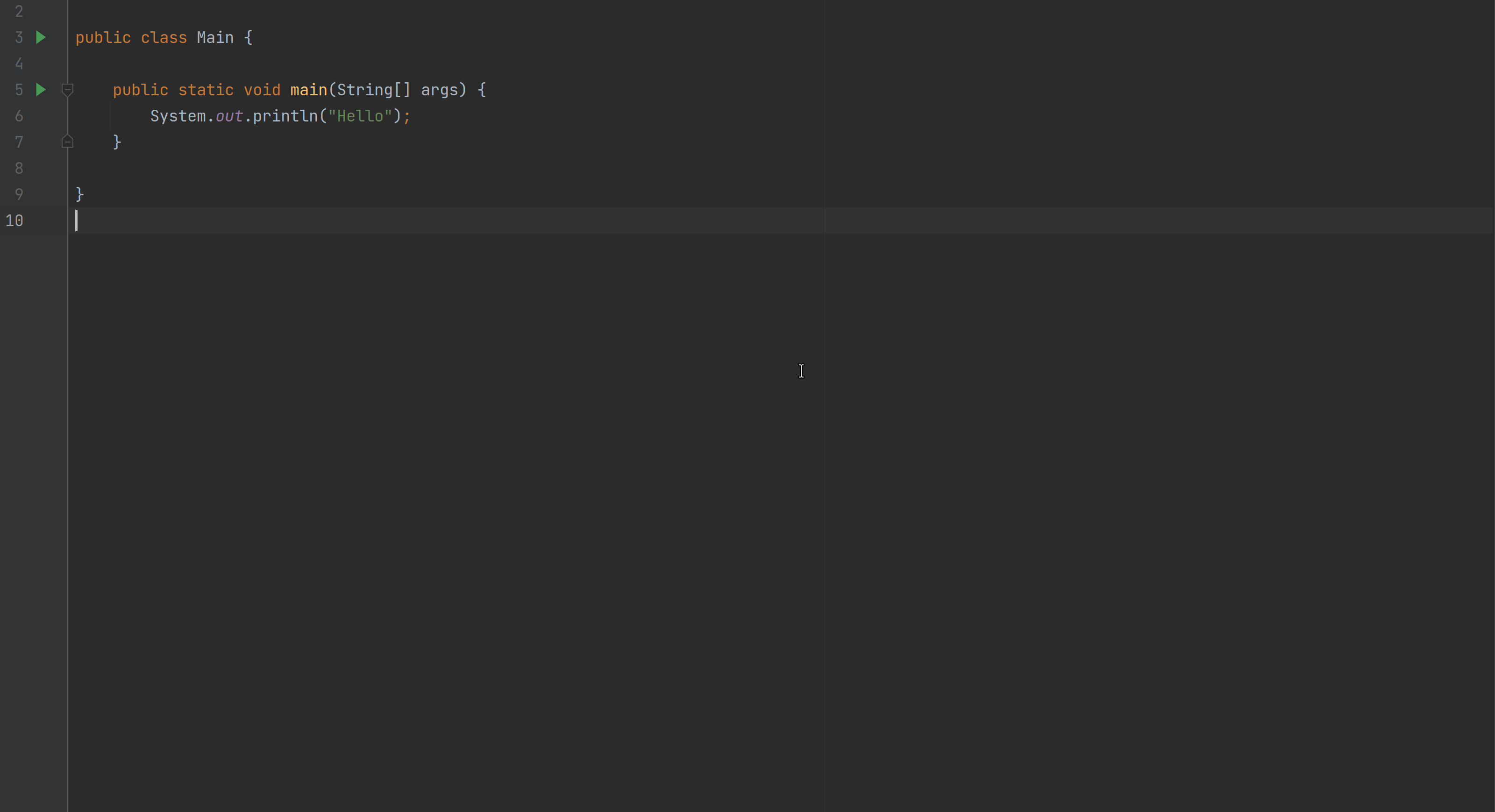
**Please note:** There are others options you could use for “compiling multiple source files in the same directory”[1]

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

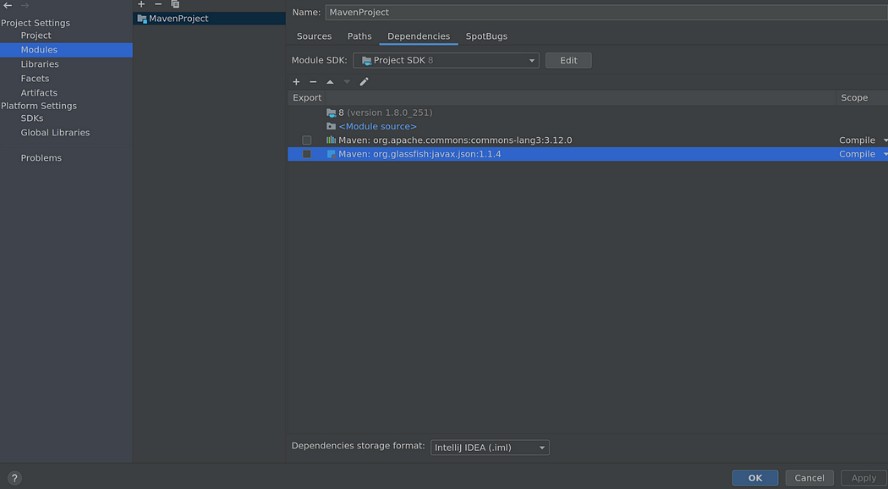
Please note: “The preferred way to specify the class path is by using the -cp command line switch”[2]

**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:

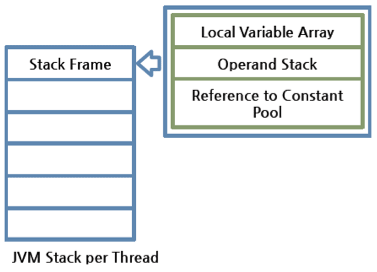
1. **Classloader reference**
2. **Run time constant pool** — Numeric constants, field references, method references, attributes; As well as the constants of each class and interface, it contains all references for methods and fields. When a method or field is referred to, the JVM searches the actual 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:**

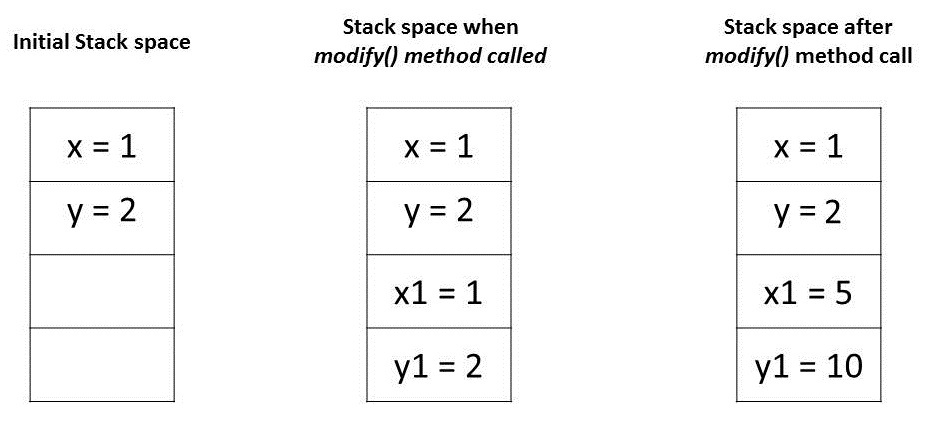
1. The Java Programming Language features **eight primitive data** types.
2. Primitive variables are directly **stored in stack memory.**
3. Whenever any variable of primitive data type is passed as an argument, the actual parameters are **copied to formal arguments** and these **formal arguments accumulate** 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:**

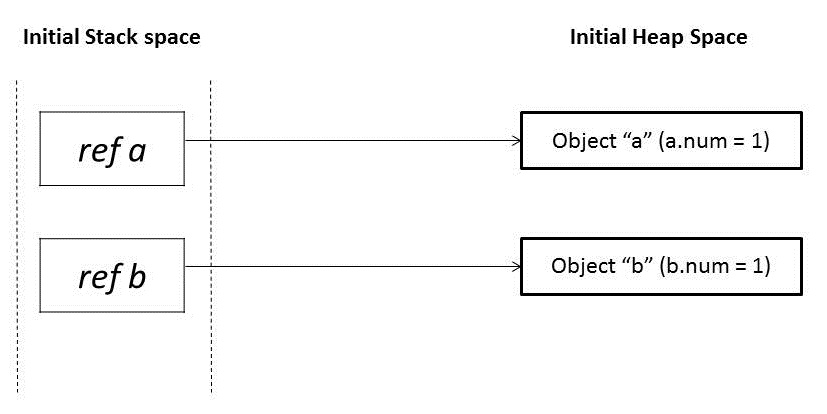
1. In Java, all objects are dynamically stored in Heap space under the hood.
2. These objects are referred from references called reference variables.
3. A Java object, in contrast to Primitives, is stored in two stages.
4. The reference variables are stored in stack memory and the object that they’re 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:

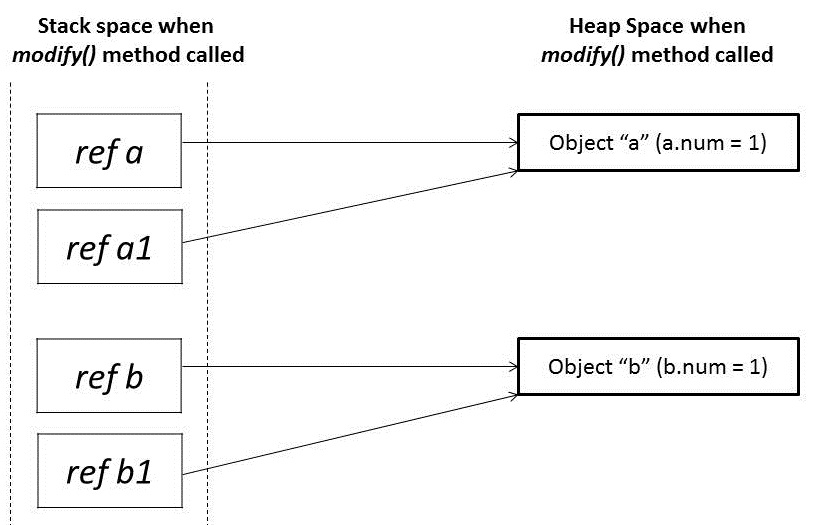
|  |
| --- |
| public class NonPrimitivesUnitTest {    @Test  public void whenModifyingObjects\_thenOriginalObjectChanged() {  Foo a = new Foo(1);  Foo b = new Foo(1);  // Before Modification  assertEquals(a.num, 1);  assertEquals(b.num, 1);    modify(a, b);    // After Modification  assertEquals(a.num, 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:**

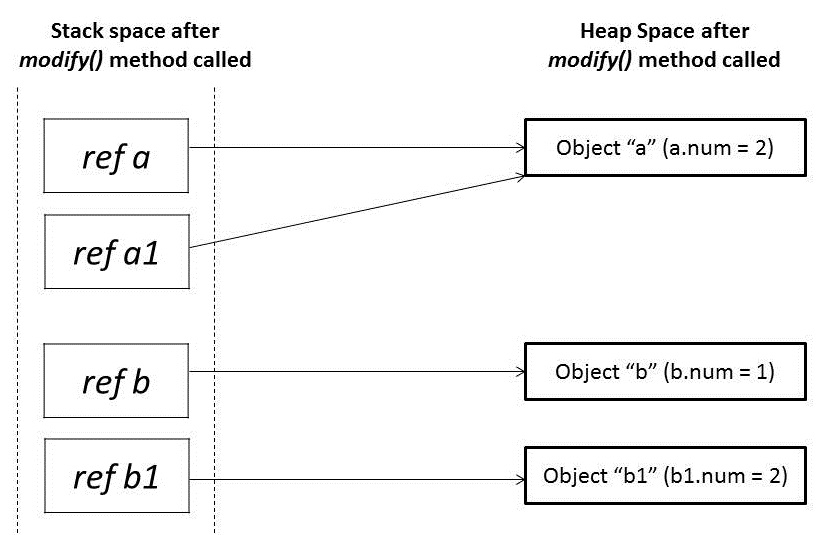
1. Let’s analyze the assertions in the above program.
2. We have passed objects a and b in modify() method that has the same value 1.
3. Initially, these object references are pointing to two distinct object locations in a heap space:



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:

1. How thread will get created in executor service framework in java internally - <https://medium.com/codex/executorservice-internal-working-in-java-7b286882f54e>
2. Everything about Java Multithreading - <https://medium.com/java-interview-revision-question-bank/everything-about-java-multithreading-be234f5ac119>
3. Following article (main url) - <https://medium.com/java-interview-revision-question-bank/everything-about-java-multithreading-be234f5ac119>

**Program, Process and Threads:**

**Program** - When we execute a program, it is loaded into the Main Memory(RAM), by the Operating System(with the concept known Paging). The Operating System’s core called Kernel then creates the process which is the running instance of a program.

**Process -** In simple words, the process is just the runtime instance of our program

**Threads** - A thread is a small independent unit of execution inside a process. Multiple threads may exist within a process, running concurrently, having some local state associated with them. There may also be a global state in a process that all the threads share. Special care needs to be taken when multiple threads are writing and reading to and from this global state.

The word thread can mean two things:

1. **Kernel Space Threads:** Threads that run in Kernel on behalf of User Space Threads: **Ex**: Device Driver Threads
2. **User Space Threads:** Threads that run in a User Level Process. **Ex**: Java Threads

**Native Thread Modelling:** Java supports (From Java 1.3 version), Native Thread Modelling with the support of underlying OS especially Linux. Linux had provided support for the large concurrent execution of threads using **the POSIX thread library**. This library is the basis for JVM implementing the Native Thread Model and provides a one-to-one mapping between Java and Kernel threads.

**Ways to Create Threads**

1. Instantiating Thread class with Runnable class

|  |
| --- |
| package learning.multithreading;  public class ThreadRunnable {  public static void main(String[] args) {  Runnable runnableTest = new RunnableTest();  Thread runnableTestThread = new Thread(runnableTest);  runnableTestThread.start();  }  }  class RunnableTest implements Runnable {  /\*\*  \* When an object implementing interface {@code Runnable} is used  \* to create a thread, starting the thread causes the object's  \* {@code run} method to be called in that separately executing  \* thread.  \* <p>  \* The general contract of the method {@code run} is that it may  \* take any action whatsoever.  \*  \* @see Thread#run()  \*/  @Override  public void run() {  ThreadInformation.print(Thread.currentThread());  }  class ThreadInformation {  public static void print(Thread thread) {  System.out.println("Current Thread Name - " + thread.getName());  System.out.println("Current Thread Priority - " + thread.getPriority());  System.out.println("Current Thread Group - " + thread.getThreadGroup());  System.out.println("Current Thread Id - " + thread.getId());  System.out.println("Current Thread State - " + thread.getState());  }  }  }  /\*\*  \* Output:  \* Current Thread Name - Thread-0  \* Current Thread Priority - 5  \* Current Thread Group - java.lang.ThreadGroup[name=main,maxpri=10]  \* Current Thread Id - 16  \* Current Thread State - RUNNABLE  \*/  // Thread group is showing from where the child thread is created.  // In the above case its main. |

1. Subclassing Thread class

|  |
| --- |
| package learning.multithreading;  public class ThreadSubclass {  public static void main(String[] args) {  Thread thread = new ThreadSubclassTest();    // Shows initial state of thread  ThreadSubclassTest.ThreadInformation.print(thread);  thread.start();  }  }  class ThreadSubclassTest extends Thread {  public void run() {  ThreadInformation.print(Thread.currentThread());  }  class ThreadInformation {  public static void print(Thread thread) {  System.out.println("Current Thread Name - " + thread.getName());  System.out.println("Current Thread Priority - " + thread.getPriority());  System.out.println("Current Thread Group - " + thread.getThreadGroup());  System.out.println("Current Thread Id - " + thread.getId());  System.out.println("Current Thread State - " + thread.getState());  }  }  }  /\*\*  \* Output:  \* Current Thread Name - Thread-0  \* Current Thread Priority - 5  \* Current Thread Group - java.lang.ThreadGroup[name=main,maxpri=10]  \* Current Thread Id - 16  \* Current Thread State - NEW  \* -----------------------------------------  \* Current Thread Name - Thread-0  \* Current Thread Priority - 5  \* Current Thread Group - java.lang.ThreadGroup[name=main,maxpri=10]  \* Current Thread Id - 16  \* Current Thread State - RUNNABLE  \*/ |

**What is the difference between Runnable and callable interface?**

**What is Callable?**

* Callable is an interface in Java that defines a single method called call(). This method is similar to the run() method of the Runnable interface, but it can return a value. Callable is primarily used to execute a task in a separate thread and retrieve the result of that task once it's completed.

Here's an example of how Callable works in Java:

|  |
| --- |
| import java.util.concurrent.Callable;  import java.util.concurrent.FutureTask;  public class MyCallable implements Callable<Integer> {  public Integer call() throws Exception {  int sum = 0;  for (int i = 1; i <= 10; i++) {  sum += i;  }  return sum;  }  public static void main(String[] args) throws Exception {  Callable<Integer> myCallable = new MyCallable();  FutureTask<Integer> futureTask = new FutureTask<>(myCallable);  Thread thread = new Thread(futureTask);  thread.start();  System.out.println(futureTask.get());  }  } |

Explanation:

* In this example, we create a class called MyCallable that implements the Callable interface. The call() method in MyCallable calculates the sum of the numbers from 1 to 10 and returns the result.
* Output of the above program is as follows – 55
* We then create an instance of MyCallable and pass it to a FutureTask object. A FutureTask is a class in Java that represents a task that will be executed in a separate thread. We then create a Thread object and pass the FutureTask object to its constructor. Finally, we start the thread and use the get() method of the FutureTask object to retrieve the result of the task.

What is Runnable?

* Runnable is another interface in Java that defines a single method called run(). The run() method is used to define the code that will be executed in a separate thread.

Here's an example of how Runnable works in Java:

|  |
| --- |
| public class MyRunnable implements Runnable {  public void run() {  for (int i = 1; i <= 10; i++) {  System.out.println(i+" ");  }  }  public static void main(String[] args) {  Runnable myRunnable = new MyRunnable();  Thread thread = new Thread(myRunnable);  thread.start();  }  } |

In this example, we create a class called MyRunnable that implements the Runnable interface. The run() method in MyRunnable prints the numbers from 1 to 10 to the console.

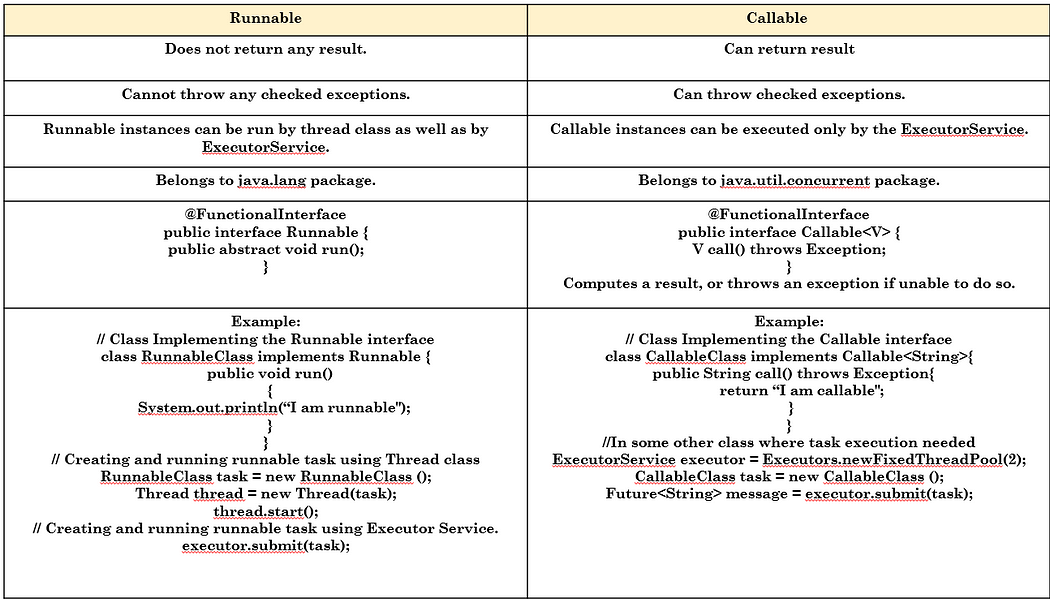
**What is Future?**

Future is a interface in Java that represents the result of a computation that will be completed in the future. A Future object can be used to check if the computation is complete, retrieve the result of the computation, or cancel the computation.

|  |
| --- |
| import java.util.concurrent.Callable;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.Future;  public class Main {  public static void main(String[] args) throws Exception {  ExecutorService executor = Executors.newSingleThreadExecutor();  Callable<Integer> myCallable = new MyCallable();  Future<Integer> future = executor.submit(myCallable);  System.out.println(future.get());  executor.shutdown();  }  } |

Explanation:

* In this example, we create an ExecutorService object using the Executors.newSingleThread
* Output of the above program is as follows -55 // same as callable, because we are calling callable class from here
* Thus, Java provides several interfaces to help developers create efficient and robust concurrent and parallel programs. The Runnable interface is used to create a simple thread, while the Callable interface is used to create a thread that returns a result. The Future interface is used to retrieve the result of a Callable interface. These interfaces are essential for building Java applications that require concurrency and multithreading capabilities.



| **Parameter** | **Runnable Interface** | **Callable Interface** |
| --- | --- | --- |
| Generics | Runnable is not a generic interface. | Callable allows us to specify the type of result that will be returned by the call() method. |
| Return value | The runnable interface has a run( ) method that does not return anything. | The Callable interface has a call( ) method that returns a value. |
| Future object | Runnable does not provide this functionality. | When a Callable is submitted to an ExecutorService, it returns a Future object that can be used to retrieve the result of the computation. |
| Exception handling | The run( ) method of the Runnable interface cannot throw a checked exception. | The call( ) method of the Callable interface can throw an exception. |
| Package | It is a part of java.lang package. | It is a part of java.util.concurrent package. |

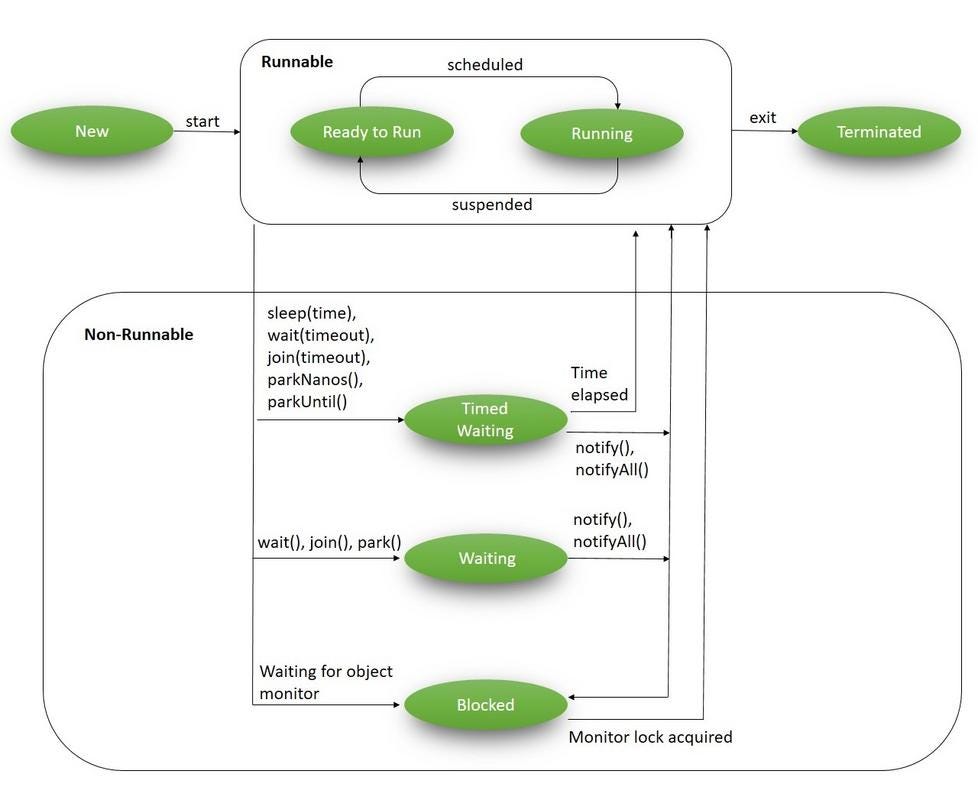
q) Can a class implement both Callable and Runnable interfaces?

**Yes**, a class can implement both Callable and Runnable interfaces. However, they serve different purposes and should be used based on the specific requirements of the task.

|  |
| --- |
| **Note:** Future is mediator between callable and Thread/any networking resource /data base resource/executor service. Future is holding callable task response and we can get the response anytime.  Future is capable of accepting both Callable and Runnable interfaces references in their own constrictors. |

**Lifecycle of a Thread**

Below diagram shows the state of Threadin its lifecycle —



**Runnable State: [Thread t = new Thread();]**

* Just thread object got created and It is in ready state.
* When you call t.start() method it will get register with Thread scheduler and it will be either Runnable(Ready to run) or Running state(If it get allowed y thread scheduler to execute it).
* **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.
* **Runnable State**: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
* A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
* terminates because of either of the following reasons:

**Non-Runnable State:**

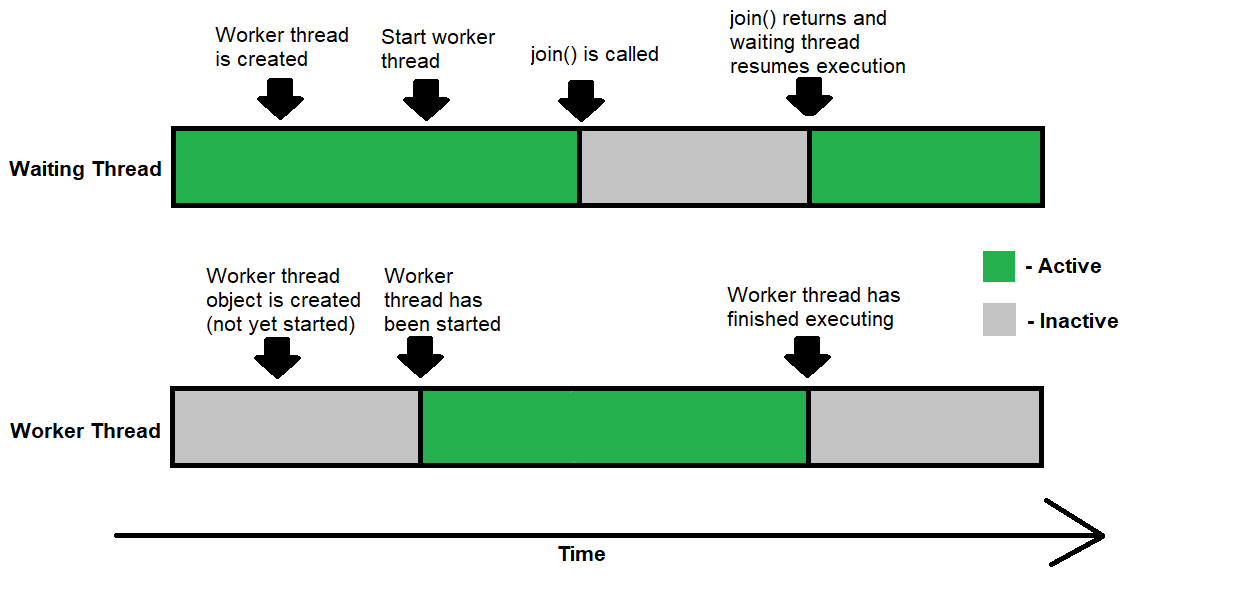
* When we interrupt running thread by calling/invoking any of the following method then it will go to Non-Runnable state.
* **Join(), wait(), yield(), notify(), notifyAll(), sleep() and park()** etc.
* **Blocked/Waiting state**: When a thread is temporarily inactive, then it’s in one of the following states — Blocked or Waiting
* **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

**To bring running thread from running state to non-runnable state, we will use below methods:**

1. **public final void join() throws InterruptedException | public final synchronized void join(final long millis) throws InterruptedException | public final synchronized void join(long millis, int nanos) throws InterruptedException**

Explanation:

* When we invoke the join() method on a thread, the **calling thread** goes into a waiting state. It remains in a waiting state until the referenced thread terminates.
* Reference link - <https://medium.com/@avinashsoni9829/java-threads-internals-join-method-part-9-dca319daac06>
* When the waiting thread calls join(), it will pause until the worker thread has terminated. No additional logic is needed in the worker thread to ensure synchronization. The waiting thread will automatically resume once the worker thread terminates. Take a look at the snippet below:
* In the snippet above, we have two threads. The waiting thread creates and starts the worker thread. It then waits, or pauses until the worker thread has finished. The waiting thread then resumes execution. See the chart below to see when each thread in the snippet above is active:
* **This approach to thread synchronization has a key limitation:** the waiting thread must wait until the other thread has terminated until it can resume! By default, the join() method does not allow the waiting and worker threads to be run simultaneously in a controlled manner. The waiting thread pauses until the worker thread either terminates or until the timeout period is reached — whichever occurs first.
* **Join() method in thread class is non-**static.



* Therefore, the join() method is a blunt instrument: it does not allow granular control of when each thread is active. join() provides no mechanism for the worker thread to resume the waiting thread programmatically. Since the worker thread must terminate for the waiting thread to resume, join() does not allow both the waiting and worker threads to run simultaneously (unless a timeout is reached).

|  |
| --- |
| //Assume have a class Worker that extends the Thread class  //Currently in the waiting thread  Thread worker = new Worker();  worker.start(); // start the worker thread  try {  worker.join(); // wait for the worker thread to finish  } catch (InterruptedException e) {  // Handle exception if necessary  }  //This line is only reached after the worker thread terminates  System.out.println("The worker thread has terminated");  //This thread now continues executing |

**Another Example**: Communicating tow threads with each other using **join()**

|  |
| --- |
| package com.example.interuptmethods;  public class JoinMethodDemo extends Thread {  public int processingCount = 0;  JoinMethodDemo(int processingCount) {  this.processingCount = processingCount;  System.out.println("Thread Created");  }  @Override  public void run() {  System.out.println("Thread " + this.getName() + " started");  while (processingCount > 0) {  try {  **Studnet s = new Studnet();**  **s.start();**  **s.join();**  Thread.sleep(1000);  } catch (InterruptedException e) {  System.out.println("Thread " + this.getName() + " interrupted");  }  processingCount--;  System.out.println("Inside Thread " + this.getName() + ", processingCount = " + processingCount);  }  System.out.println("Thread " + this.getName() + " exiting");  }      public static void main(String[] args) throws InterruptedException {    **Thread t1 = new Thread(new Studnet());**  **t1.start();**    System.out.println("Returned from join");  }    }  class Studnet extends Thread{  @Override  public void run() {  **JoinMethodDemo j = new JoinMethodDemo(1);**  **j.start();**  **try {**  **j.join();**  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  System.out.println("Student thred run method");  }    }  **Output:**  Thread Thread-147036 started  Thread Created  Thread Thread-147038 started  Thread Created  Thread Thread-147040 started  Thread Created  Thread Thread-147042 started  #  # There is insufficient memory for the Java Runtime Environment to continue.  # Native memory allocation (malloc) failed to allocate 1176448 bytes for AllocateHeap  # An error report file with more information is saved as:  # D:\Hari -WS\Core-Java-MultiThreading-FileConcepts-WS\07-corejava-multithreading-medium-article\hs\_err\_pid128468.log |

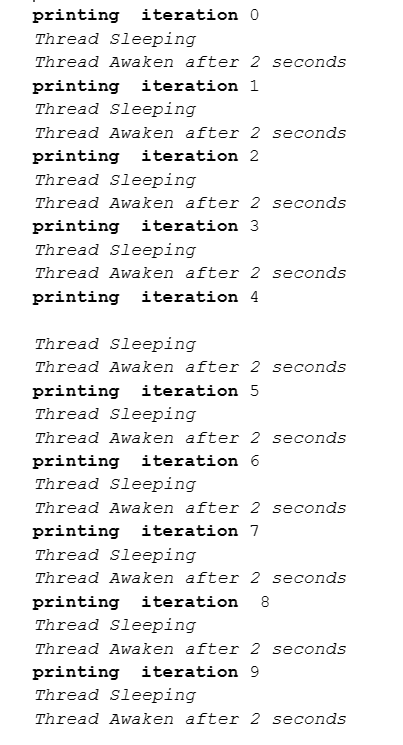
1. **public static void sleep(long millis) throws InterruptedException | public static void sleep(long millis, int nanos) throws InterruptedException | public static void sleep(Duration duration) throws InterruptedException**

**Explanation:**

* Sleep() method in Thread class is tstaic method.
* Sleep- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
* The thread does not lose ownership of any monitors.

|  |
| --- |
| for(int i = 0 ; i < 10 ; ++i)  {  System.out.println(" printing iteration " + i);  try {  System.out.println("Thread Sleeping ");  Thread.sleep(1999);  System.out.println("Thread Awaken after 2 seconds ");  }  catch (InterruptedException e)  {  e.printStackTrace();  }  } |

Output:



**Example:** Let’s Also See How interrupts Work Using Sleep as an Example

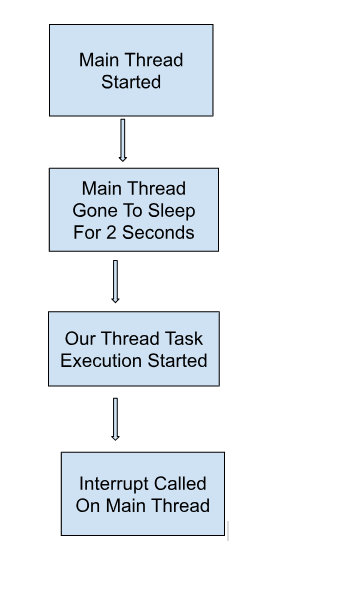
|  |
| --- |
| public class InterruptHandler implements Runnable {  @Override  public void run() {  try {  System.out.println(Thread.currentThread().getName() + " : " + "Task Started");  System.out.println(Thread.currentThread().getName() + " : "+ "Sleeping the thread for 5 seconds");  Thread.sleep(5000);  System.out.println("Task Ended");  }  catch (InterruptedException e)  {  e.printStackTrace();  }  }  public static void main(String[] args) {  Thread thread = new Thread(new InterruptHandler());  thread.start();  thread.setName("MyThread");  try {  System.out.println(Thread.currentThread().getName() + " : " + "Sleeping the thread for 2 seconds " );  Thread.sleep(2000);  System.out.println(Thread.currentThread().getName() + " : " + "Calling Interrupt");  thread.interrupt();  }  catch (InterruptedException e)  {  System.out.println(Thread.currentThread().getName() + " : " + "Printing Stack Trace ");  e.printStackTrace();  }  }  } |

**Note :**

* InterruptHandler Class Run Method Makes the Thread Sleep for 5 s
* The interrupt method is called after 2s of its running ( currently it is in sleep only)

**Terminated State:**

* A thread terminates because of either of the following reasons:
* Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
* Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.



**Java Thread interrupt() method**

* The interrupt() method of thread class is used to interrupt the thread. **If any thread is in sleeping or waiting state (i.e. sleep() or wait() or join(), join(long), join(long, int), sleep(long), or sleep(long, int) is invoked) then using the interrupt() method**, we can interrupt the thread execution by throwing InterruptedException.
* **If the thread is not in the sleeping or waiting state then calling the interrupt() method performs a normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.**
* **Syntax** : public void interrupt()
* **Return**: This method does not return any value.
* **Exception** : SecurityException: This exception throws if the current thread cannot modify the thread.

**Example1:**

|  |
| --- |
| package com.example.interuptmethods;  public class ThreadsInJava  {  public static void main(String[] args)  {  Thread t = new Thread()  {  public void run()  {  System.out.println("First line of run method " + isInterrupted()); //Output : true    try  {  System.out.println("Before sleep");  Thread.sleep(10000); //Thread is going to sleep for 10 seconds  System.out.println("After sleep");  }  catch (InterruptedException e)  {  System.out.println("Thread is interrupted");  }    System.out.println("Last line of run method" + isInterrupted()); //Output : false  }  };    t.start();    t.interrupt(); //main thread is interrupting thread t    System.out.println("Is alive method runnig by main thread" + t.isAlive());    System.out.println("Main thread is running");  }  } |

Exception: SecurityException: This exception throws if the current thread cannot modify the thread.

**Explanation:**

* Intrupt() method can interrupt a thread which is in Non-runnable[waiting, sleeping, join stage] then it will throw the interrupt exception and it will terminate the stage of thread life.
* If you try to print t.isAlive after interrupt() method, you will see false
* If you try to interrupt() thread which is in running state it will not get interrupt, but it will set the interrupt flag as true and when thread goes to sleep state, then it will throw the exception.

1. **public final void join() throws InterruptedException | public final synchronized void join(final long millis) throws InterruptedException | public final synchronized void join(long millis, int nanos) throws InterruptedException**

**Executor Framework in Java:**

* The Executor Framework is a powerful and flexible tool for managing and executing tasks in Java applications. It provides a way to separate the task execution logic from the application code, allowing developers to focus on business logic rather than thread management.
* The framework includes several key components, including the **Executor, ExecutorService, ScheduledExecutorService, and ThreadPoolExecutor**. These components enable developers to control the number of threads, manage task priorities, and handle exceptions and timeouts.
* The Executor Framework is particularly useful for **managing concurrent tasks** in applications with a **large number of threads** or **high levels of concurrency**.
* It is widely used in applications such as **web servers, where multiple requests** must be processed simultaneously.
* By providing a simple and efficient way to manage task execution, the Executor Framework can help developers improve the performance and scalability of their applications while minimizing the risk of threading errors and other concurrency issues.

**Executor:**

* This interface provides a way to execute submitted Runnable tasks.
* An Executor is normally used instead of explicitly creating threads.
* For example, rather than invoking new Thread(new RunnableTask()).start() for each task of a set of tasks, you might use:

|  |
| --- |
| Executor executor = new ThreadPoolExecutor(1, 10,  0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>());  executor.execute(runnableTask); |

* **execute():** This execute() method executes the given Runnable task at some time in the future. The command(task) may execute in a new thread, from the pooled thread, or in the calling thread, at the discretion of the Executor implementation (e.g. ThreadPoolExecutor).
* In the above sample code runnableTask is a task created using lambda implementation of the **run()** method of the Runnable interface.

|  |
| --- |
| Runnable runnableTask = () -> {  try {  System.out.println("Run method called.");  TimeUnit.MILLISECONDS.sleep(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }; |

* The Executor interface is part **of java.util.concurrent** package and was introduced in **Java 1.5.**

**ExecutorService:**

* An ExecutorService provides methods to manage termination and methods that can produce a Future for tracking the progress of one or more asynchronous tasks.
* An ExecutorService can be shut down, which will cause it to reject new tasks.

|  |
| --- |
| **ExecutorService is an extended version of Executor with more methods and features.** |

What is Object oriented programming:

To achieve logical operation/calculation is called programming and object contains data or object stores the data.

Object has three main properties

1. State: data, variable and related properties
2. Behavior: behavior - to invoke the methods
3. Identity: to create address (id=27) – to JVM purpose

**Locking Mechanism in java:**

We have two type of locking mechanism in java:

1. Object level locking
2. Class level locking

**Object-Level Locking**

* In the parallel universe of multithreaded programming, chaos reigns supreme without proper synchronization.
* Object-level locking in Java orchestrates the cacophony, ensuring that threads march to the same rhythm.

|  |
| --- |
| An object-level lock is a mechanism when we want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on a given instance of the class. If a thread wants to execute a synchronized method on the given object. |

**Important Points:**

1. **Every object in java has a unique lock.**
2. Whenever we are using a synchronized keyword, then only the lock concept will come into the picture.
3. If a thread wants to execute a synchronized method on the given object.
4. First, it has to get a lock of that object.
5. Once the thread got the lock then it is allowed to execute any synchronized method on that object.
6. Once method execution completes automatically thread releases the lock.
7. Acquiring and release lock internally is taken care by JVM and the programmer is not responsible for these activities

**Methods:** There are different ways we can lock the object in the thread as below:

1. Method 1: By applying synchronized keyword to non-static method

|  |
| --- |
| public class GeekClass  {  public **synchronized** void GeekMethod(){}  } |

1. Method 2: By applying synchronized keyword to non-static block by giving this(current object) as reference to acquire the lock

|  |
| --- |
| public class GeekClass  {  public void GeekMethod(){  synchronized (this)  {  // other thread safe code  }  }  } |

1. Method 3: By applying synchronized keyword to non-static block by giving another instance object as lock acquiring object.

|  |
| --- |
| public class DemoClass  {  private final **Object lock** = new Object();  public void demoMethod(){  synchronized (lock)  {  // other thread safe code  }  }  } |

Types of Object-Level Locks - Object-level locks in Java can be manifested in two primary forms:

1. **Synchronized Methods:** Using the synchronized keyword with a method.
2. **Synchronized Blocks**: Using the synchronized keyword with a specific block of code.

**Synchronized Methods**

A synchronized method allows only one thread to execute the method at a time on the same object.

**Example**: A Ticket Booking System

* Imagine a movie theater where only one ticket can be booked at a time for a particular seat.

|  |
| --- |
| public class TicketBooking {  private int seatsAvailable = 10;  public synchronized void bookTicket() {  if (seatsAvailable > 0) {  seatsAvailable--;  System.out.println("Ticket booked successfully!");  } else {  System.out.println("No seats available!");  }  }  }  **How It Works:** When a thread calls bookTicket, it acquires an intrinsic lock on the object, preventing other threads from accessing the synchronized method on the same object simultaneously. |

Example 2:

|  |
| --- |
| package com.example.locks;  public class SynchronizedBlcokOnclassLevel implements Runnable {    public static Integer id = 2;    public Integer value;    // Method of this class  public void run() { Lock(); }    // Synchronization of non-static methods  // (object lock) as different synchronized  // non-static methods are called in both threads    // Then both threads need to acquire the object lock  // After one is acquired, the other thread must wait  // for one thread to finish the executing  // before the other thread starts to execute.  public void Lock()  {  System.out.println(  Thread.currentThread().getName());  //t1.start ----> g1 locked -----  //t2.start() ---> g1  //t3.start() ---> g2 --- skeliton    synchronized (this)  {  id = 3;  System.out.println(  "in block "  + Thread.currentThread().getName());  System.out.println(  "in block "  + Thread.currentThread().getName()  + " end");  System.out.println("ID value "+ id++);  }  }    // Main driver method  public static void main(String[] args)  {  // Creating an object of above class  // in the main() method  SynchronizedBlcokOnclassLevel g1 = new SynchronizedBlcokOnclassLevel();    SynchronizedBlcokOnclassLevel g2 = new SynchronizedBlcokOnclassLevel();    // Sharing the same object across two Threads    // Here, t1 takes g  Thread t1 = new Thread(g1);    // Here, t2 takes g  Thread t2 = new Thread(g2);    // Creating another object of above class  SynchronizedBlcokOnclassLevel g3 = new SynchronizedBlcokOnclassLevel();    // Here, t3 takes g1  Thread t3 = new Thread(g3);    // setname() method is used to change  // name of the thread  t1.setName("t1");  t2.setName("t2");  t3.setName("t3");    // start() method beginning the execution of threads  // as JVM calls the run() method of thread  t1.start();  t2.start();  t3.start();  }  } |

**Intrinsic lock vs Extrinsic lock:**

* In Java 5.0, a new addition called Reentrant Lock was made to enhance intrinsic locking capabilities.
* Prior to this, "synchronized" and "volatile" were the means for achieving concurrency.

|  |
| --- |
| public synchronized void doAtomicTransfer(){  //enter synchronized block , acquire lock over this object.  operation1()  operation2();  } // exiting synchronized block, release lock over this object. |

* Synchronized **uses intrinsic locks or** monitors.
* Every object in Java has an intrinsic lock associated with it.
* Whenever a thread tries to access a synchronized block or method, it acquires the intrinsic lock or the monitor on that object.

In case of static methods, the thread acquires the lock over the class object.

* An intrinsic locking mechanism is a clean approach in terms of writing code, and is pretty good for most of the use-cases.

So why do we need the additional feature of explicit locks? Let's discuss.

An intrinsic locking mechanism can have some functional limitations, such as:

1.) It is not possible to interrupt a thread waiting to acquire a lock (lock Interruptibly).

2.) It is not possible to attempt to acquire a lock without being willing to wait for it forever (try lock).

3.) Cannot implement non-block-structured locking disciplines, as intrinsic locks must be released in the same block in which they are acquired.

|  |
| --- |
| public void transferMoneyWithSync(Account fromAccount, Account toAccount,  float amount) throws InsufficientAmountException {  synchronized (fromAccount) {  // acquired lock on fromAccount Object  synchronized (toAccount) {  // acquired lock on toAccount Object  if (amount > fromAccount.getCurrentAmount()) {  throw new InsufficientAmountException(  "Insufficient Balance");  } else {  fromAccount.debit(amount);  toAccount.credit(amount);  }  }  }  } |

**Intrinsic/Monitor Vs Reentrant Lock in Java:**

Java programming language supports multithreading or concurrency very well. Therefore, when we say multithreading or concurrent programming, it is necessary to have efficient synchronization or locking mechanisms, which saves concurrent program from unexpected concurrency issues like deadlocks. Reentrancy is one of those locking mechanism which help concurrent program to work as expected without any locking issue. Now, you want to know, what exactly this reentrancy do in Java? Let me explain you reentrancy or reentrant locking in Java, but before moving on it, I would like to tell you about, how Java threads take locks on objects and make operation synchronize or atomic. Therefore, before moving on reentrancy, I would like to explain intrinsic or monitor lock.

Intrinsic Or Monitor Lock Java provide built-in locking mechanism to make any operation atomic by synchronization in concurrent environment. In Java programming language ‘synchronized' keyword is used for taking lock or monitor on object. There are other locking utility also available in Java but we will concentrate on basic locking mechanism (synchronized) only. synchronized keyword can be use with method signature or with particular code block to make that synchronize. These built-in locks known as intrinsic or monitor lock.

Intrinsic Lock Examples There are few intrinsic or monitor lock examples mentioned below to make code synchronized.

1. Creating object level lock with method signature.

|  |
| --- |
| public synchronized void objectLevelLock() {  // Code to make synchronized or atomic  } |

2. Creating object level lock with synchronized block inside method.

|  |
| --- |
| public void objectLevelLock () {  synchronized(lockObject) {  // Code to make synchronized or atomic  }  } |

3. Creating class level lock on static members of class with method signature

|  |
| --- |
| public static synchronized void classLevelLock () {  // Static members and Code to make synchronized or atomic  } |

4. Creating class level lock on static members of class with synchronized block inside method.

|  |
| --- |
| public void classLevelLock () {  synchronized(ClassName.class) {  // Static members and Code to make synchronized or atomic  }  } |

Reentrant Lock Reentrancy is a locking mechanism provide by Java, which prevent Java locking from critical concurrency issue like deadlock . Suppose, there are two threads A and B. Thread B is trying to acquire lock or monitor on an object, which is already acquired by thread A. Thread B will be blocked until thread A release lock or monitor. But if thread A will try to acquire lock on other synchronized method or block on same object, it will be succeeded to acquire lock on same object. This facility is known as reentrancy. In other words, a thread who takes lock or monitor on object can be reenter any number of synchronized methods or blocks of same object on which it has already acquired lock. This is because, object locking is performed on per thread basis, not on per invocation basis.

What Could Be The Problem Without Reentrant Lock? In this section I would like to explain, If reentrancy wouldn't supported by Java, how it could affect thread execution and create concurrency issue. Please carefully go through the code given below.

|  |
| --- |
| public class Reentrancy {  public synchronized void inner() {  // Code to make synchronized or atomic  }  public synchronized void outer() {  // Code to make synchronized or atomic  inner();  }  } |

Reference link: <https://wiserhawk.blogspot.com/2017/01/intrinsic-vs-reentrant-lock-in-java.html> && <https://dzone.com/articles/what-are-reentrant-locks>

In this above code what will happen if reentrancy is not supported by Java. Suppose, a thread acquire lock on Reentrancy class object by invoking synchronized method outer(). But if you would notice outer() method invoking inner() method, which is also synchronized method. In such case, if Java wouldn’t support reentrancy, outer() method couldn’t succeeded to acquire lock on inner() method, because, it would be considered already locked and eventually result would be deadlock. But just because reentrancy is supported in Java, the thread who acquire lock on object can enter any synchronized method or block on same object to acquire lock.

**How Reentrant Lock Works In Java?** Reentrancy is implemented in Java by associating a counter with each thread. Initially counter initialized with value zero and considered as unlocked. When thread acquires lock on an object, counter get incremented by one. Again, if thread acquires lock on another synchronized method or block, counter again get incremented by one; counter will become two and so on. Same reverse process is followed, when thread release lock by leaving synchronized method or block. When thread releases lock from synchronized method or block; counter get decremented by one and so on. Once again, when counter reached to zero; object gets unlocked. Now other threads are free to acquire lock on that object. This is the approach by which Java manage reentrancy.

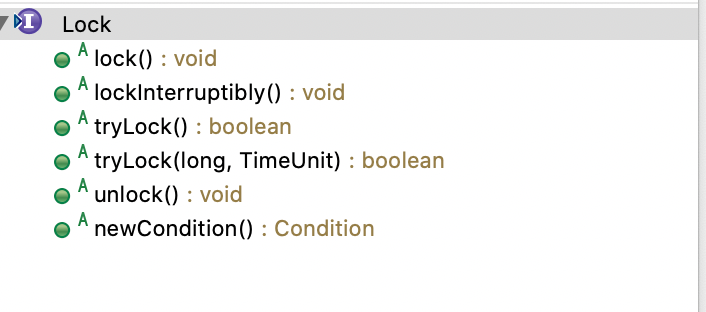
**Extrinsic locks in JAVA**

Java dedicates a complete package(java.util.concurrent.locks) for extrinsic locks. Let’s explore how these locks work under the hood.

An extrinsic lock in java must implement the java.util.concurrent.locks.Lock interface.

To add further a lock can be shared with multiple threads as well, if the implementation of the Lock interface permits it. So a lock can be Exclusive or Shared.Typically a shared lock can be used while reading (read lock) a shared resource(“critical section”).

Let’s have a brief overview of the methods in the lock interface.



* lock(): In order to acquire a lock, a thread must invoke the lock() method on the Lock Object (For Instance :Reentrant lock ) and success for acquisition depends upon the following scenario.
* In Case of a Reentrant lock, a thread can successfully acquire a lock, if the lock is not yet acquired by other thread or the thread trying to acquire the lock already holds it (Reentrant scenario). If both the scenario fails then the thread will be blocked / dormant until it gets the lock.
* lockInterruptibly() : It is similar to lock method, additionally it supports interruption. Whenever an interrupt() method is invoked on this thread then it will exit the “critical section” if it has already acquired the lock. If not, it will be in the AQS(AbstractQueuedSynchronizer) queue and the status bit of the node will be set to cancelled. We will have a separate article for AQS in this series.
* tryLock(): This method will return true, if the lock is available and acquires it, if not it returns false immediately. tryLock() method will not block the thread unlike lock() method.
* tryLock(long,TimeUnit): It is overloaded method of tryLock() method with an extended capability to wait for the specified time(argument).
* unlock() : This method releases the lock acquired via lock() method.
* newCondition(): This method returns an instance of Condition object, a condition object makes the thread to be suspend on invoking the await() method and it will be released on invoking the signal()/signalAll() method.

Example:



Reference links:

* <https://medium.com/@fullstacktips/object-level-and-class-level-locking-in-java-917ceeb9715c>
* <https://medium.com/analytics-vidhya/understanding-java-thread-synchronization-with-methods-vs-objects-vs-locks-5428e3342fee>

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?

Q) Overloading vs overriding? In terms of exception handling?

Q) Collections – List, Set, Queue (internal working)

- Map internal working

Q) Atomic Integer and ThreadLocal advantages?