**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!

**Declarative vs. Imperative Programming**

Reference Link - <https://codefresh.io/learn/infrastructure-as-code/declarative-vs-imperative-programming-4-key-differences/#:~:text=In%20declarative%20programming%2C%20state%20is,hand%2C%20requires%20explicit%20state%20management>.

**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**

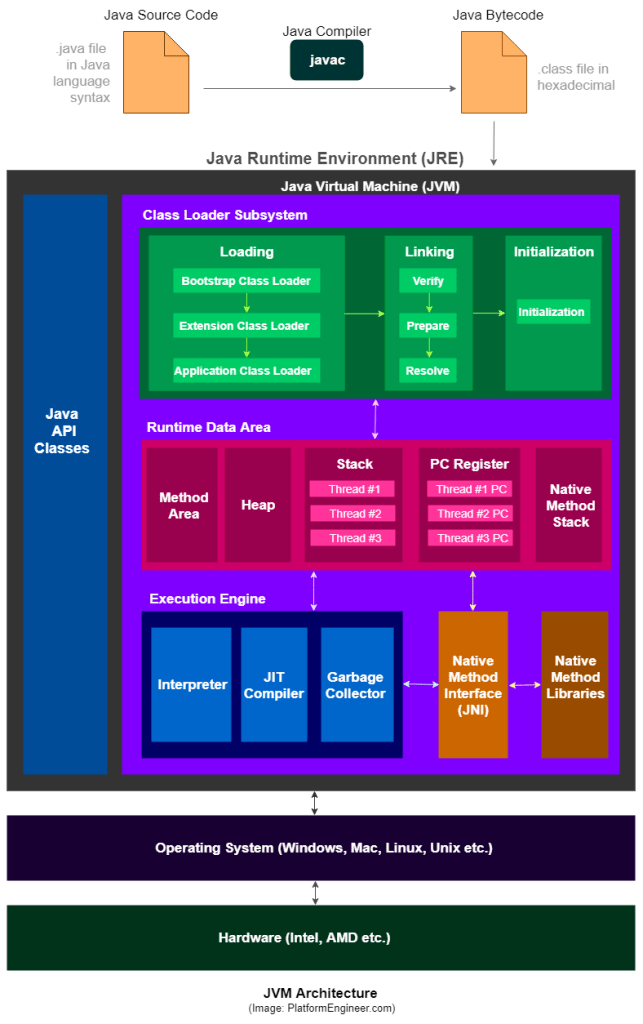
You can start writing a simple Java program with any terminal editor (vim, nano) or GUI editor (gedit, sublime). For a complex Java application, you may need an IDE (Integrated Development Environment) like In**telliJ IDEA, Eclipse, or Netbeans. A** typical Java program should contain correct language 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 –** [**https://sahiladhav16.medium.com/understanding-jvm-architecture-cc3fcee39b9a**](https://sahiladhav16.medium.com/understanding-jvm-architecture-cc3fcee39b9a)

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**

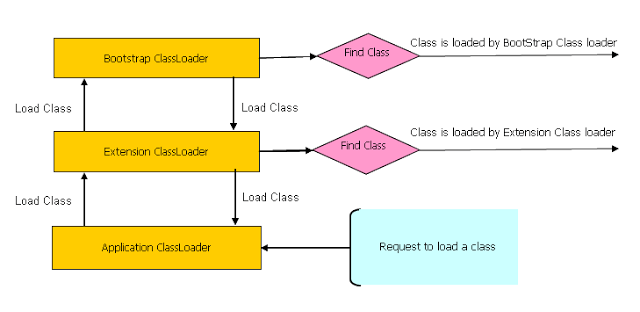
This principle states that a class loaded **by parent should not be** **loaded by Child Class Loader** **again** and ensure that duplicate class loading does not occur.

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 par hwo to generate jar file for java project with full source code inclusing system classes and libs ent, 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>

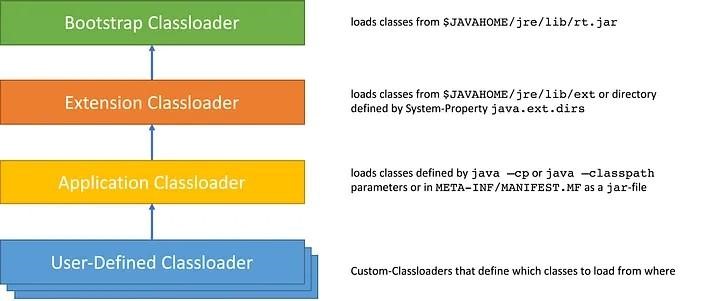
|  |
| --- |
| import java.sql.DriverManager;  import java.util.ArrayList;  public class PrintClassLoaders {  public static void main(String args[]) {  System.out.println("java.ext.dirs: "  + System.getProperty("java.ext.dirs"));  System.out.println("Classloader of ArrayList: "  + ArrayList.class.getClassLoader());  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)** Why ClassLoader delegates the search for the class or resource to its parent? What is the purpose or the advantage of doing so?

There are several valid reasons for **classloading delegation. I** have listed them according to their priority(from my understanding):

**Security-** There are certain classes in Java which shouldn't be messed with. By having parent-first delegation model, JVM can be sure that it is executing only those classes and not the one loaded by custom classloaders.

**Avoid duplicate class instances -** Classloading is a costly operation as it requires reading data from external storage or network, parsing the bytecode, allocating memory and so on. So restricting the JVM to load the classes only once is one of the reasons. By forming classloader hierarchy with the parent-first rule will achieve this.

**Class Scope -**There are certain classes which are the core part of Java, like java.lang.\*. These classes are part of Java language and will be used at almost all the places. Since a class is identified uniquely by its fully qualified name along with the classloader which loaded that class, it is important to have a single classloader to load such classes. Thus the bootstrap and extension classloader take care of this. Also, by loading the classes and resources at top level gives them a broader scope than loading at the bottom of the classloader hierarchy.

Another example - <https://stackoverflow.com/questions/51713119/class-loaders-delegation-hierarchy-algorithm>

Q) How to break singleton 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

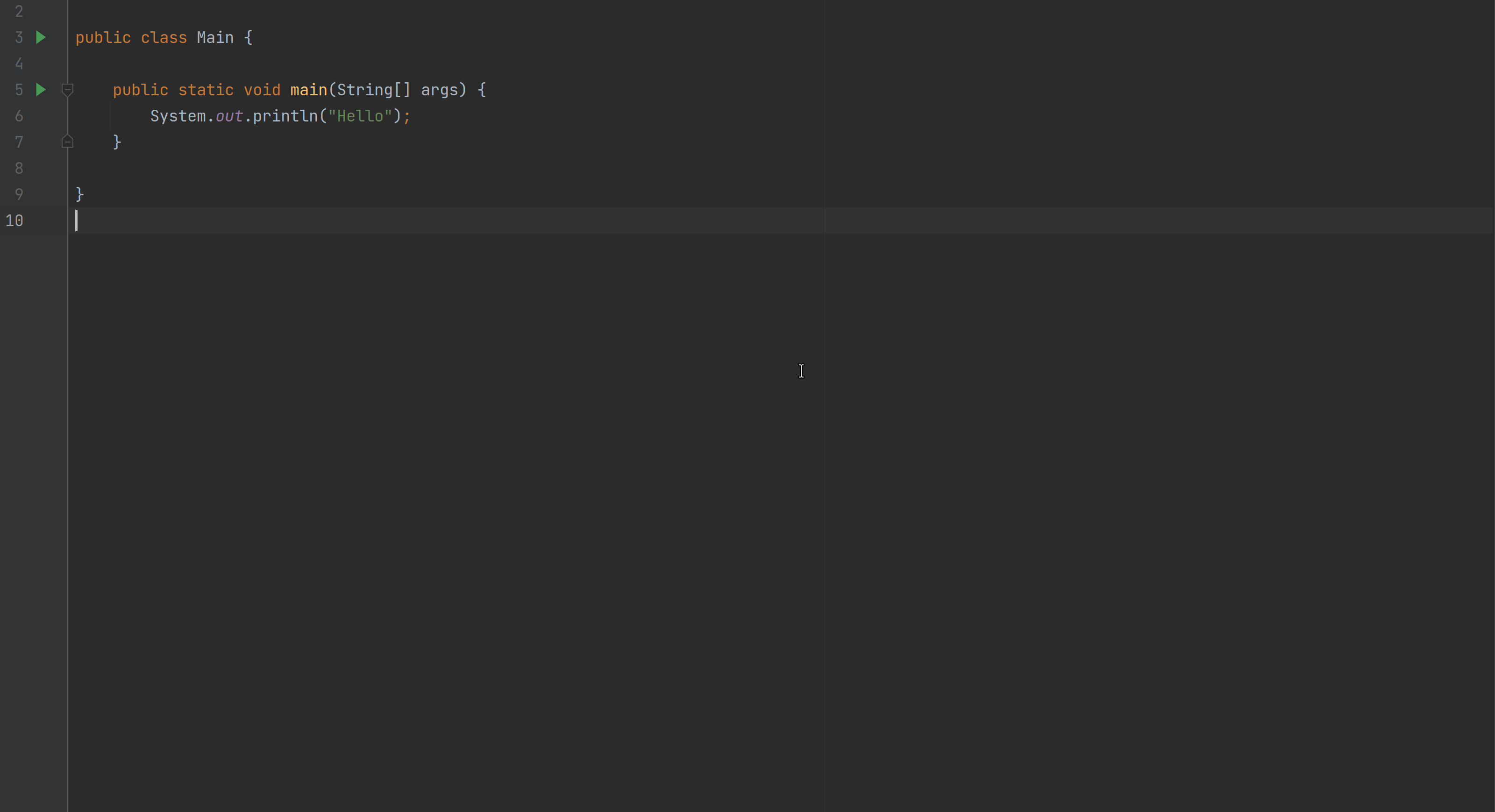
**Please note:** There are others options you could use for “compiling multiple source files in the same directory”[1]

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

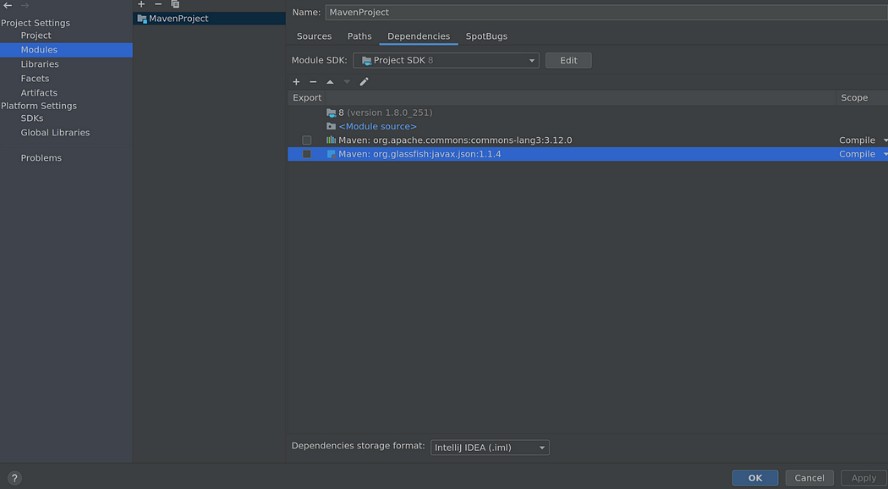
Please note: “The preferred way to specify the class path is by using the -cp command line switch”[2]

**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:

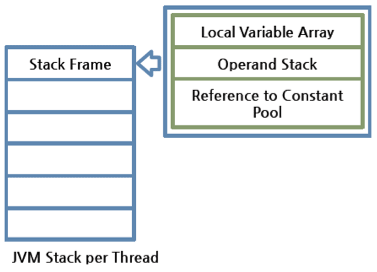
1. **Classloader reference**
2. **Run time constant pool** — Numeric constants, field references, method references, attributes; As well as the constants of each class and interface, it contains all references for methods and fields. When a method or field is referred to, the JVM searches the actual 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/machine understand(OS) is stored in **the cache**, thus the compiled code can be executed quicker.

**Important Points:**

* However, even for JIT compiler, it takes more time for compiling than for the interpreter to interpret.
* For a code segment that executes just once, it is better to interpret it instead of compiling.
* Also the native code is stored in the cache, which is an expensive resource.
* With these circumstances, JIT compiler internally checks the frequency of each method call and decides to compile each only when the selected method has occurred more **than a certain level of times.**
* This idea of adaptive compiling has been used in Oracle Hotspot VMs

Execution Engine qualifies to become a key subsystem when introducing performance optimizations by JVM vendors. Following 4 components can largely improve its performance.

* Intermediate Code Generator produces intermediate code.
* **Code Optimizer** is responsible for optimizing the intermediate codegenerated above.
* **Target Code Generator** is responsible for generating Native Code (i.e.Machine Code).

**IBM AOT (Ahead-Of-Time) Compiling**

* The specialty here is that these JVMs share the **native code compiled through** the shared cache, thus the code that has been already compiled through the AOT compiler can be used by another JVM without compiling.
* In addition, IBM JVM provides a fast way of execution by pre-compiling code to JXE (Java Executable) file format using the AOT compiler.

3.3) Garbage Collector (GC)

* As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.
* In general, garbage collection happens under the hood, however we can trigger it by calling **System.gc()** method (Again the execution is not guaranteed. Hence, call Thread.sleep(1000) and wait for GC to complete).

3.4) Java Native Interface (JNI)

* This interface is used to interact with Native Method Libraries required for the execution and provide the capabilities of such Native Libraries (often written in C/C++).
* This enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

3.5) Native Method Libraries

* This is a collection of C/C++ Native Libraries which is required for the Execution Engine and can be accessed through the provided Native Interface.

3.6) JVM Threads

* We discussed on how a Java program gets executed, but didn’t specifically mention about the executors. Actually to perform each task we discussed earlier, the JVM concurrently runs multiple threads.
* Some of these threads carry the programming logic and are created by the program (**application threads),** while the rest is created by JVM itself to undertake background tasks in the system (**system threads**).

The major system threads are as follows.

* **Compiler threads:** At runtime, compilation of bytecode to native code is undertaken by these threads.
* **GC threads**: All the GC related activities are carried out by these threads.
* **Periodic task thread:** The timer events (i.e. interrupts) to schedule execution of periodic operations are performed by this thread. Signal dispatcher thread: This thread receives signals sent to the JVM process and handle them inside the JVM by calling the appropriate JVM methods.
* **VM thread**: As a pre-condition, some operations need the JVM to arrive at a safe point where modifications to the Heap area does no longer happen. Examples for such scenarios are “stop-the-world” garbage collections, thread stack dumps, thread suspension and biased locking revocation. These operations can be performed on a special thread called VM thread

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:**

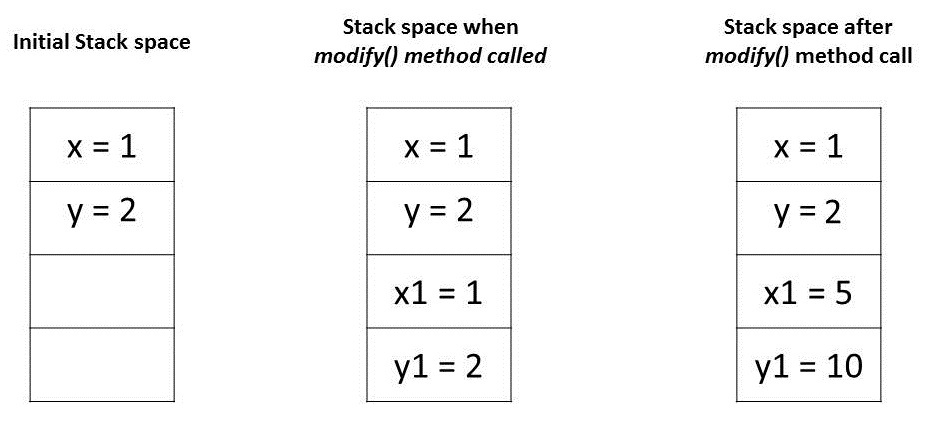
1. The Java Programming Language features **eight primitive data** types.
2. Primitive variables are directly **stored in stack memory.**
3. Whenever any variable of primitive data type is passed as an argument, the actual parameters are **copied to formal arguments** and these **formal arguments accumulate**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

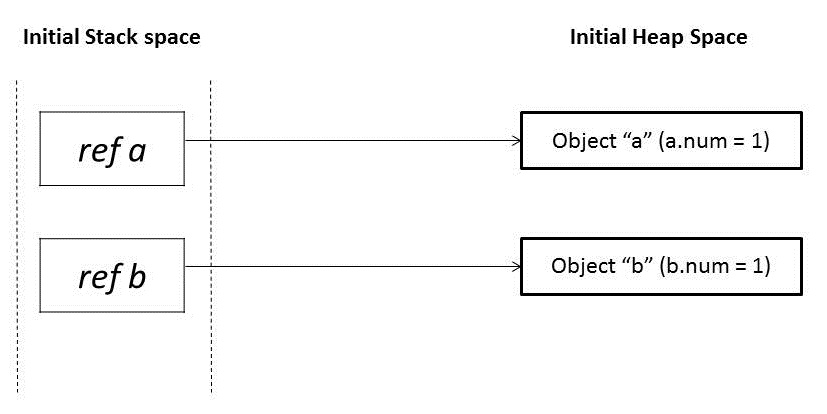
1. In Java, all objects are dynamically stored in Heap space under the hood.
2. These objects are referred from references called reference variables.
3. A Java object, in contrast to Primitives, is stored in two stages.
4. The reference variables are stored in stack memory and the object that they’re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

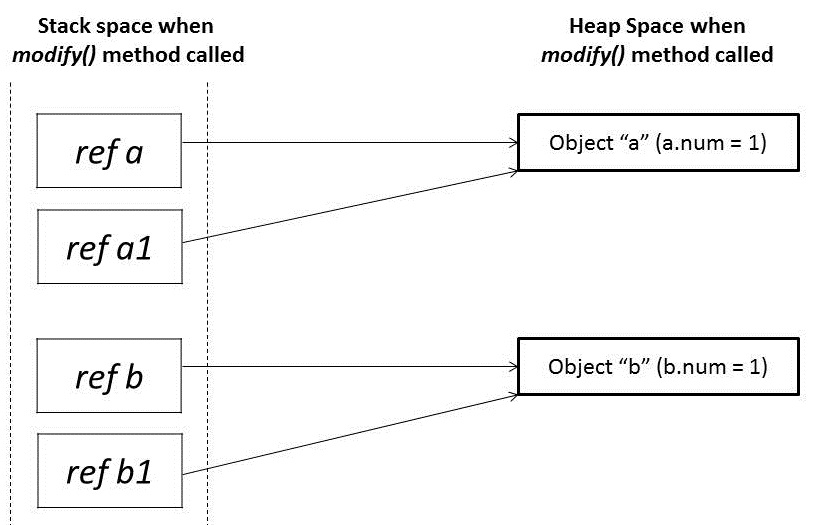
|  |
| --- |
| public class NonPrimitivesUnitTest {    @Test  public void whenModifyingObjects\_thenOriginalObjectChanged() {  Foo a = new Foo(1);  Foo b = new Foo(1);  // Before Modification  assertEquals(a.num, 1);  assertEquals(b.num, 1);    modify(a, b);    // After Modification  assertEquals(a.num,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

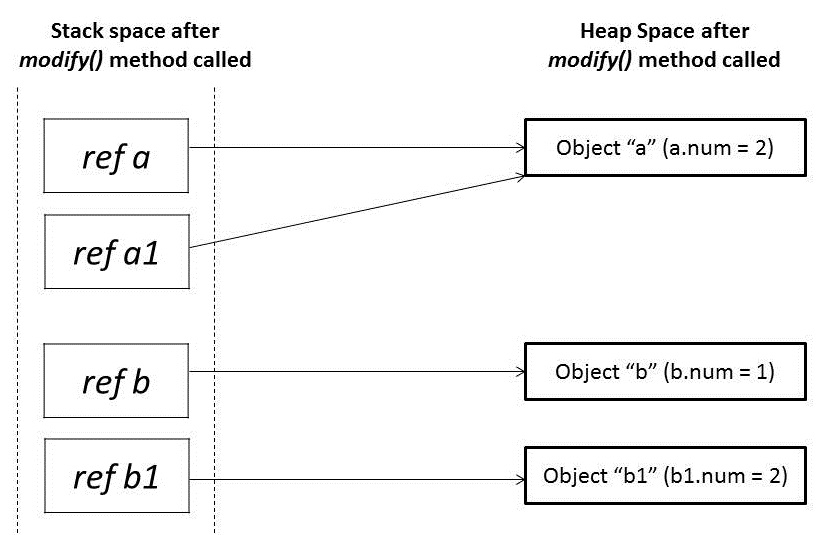
1. Let’s analyze the assertions in the above program.
2. We have passed objects a and b in modify() method that has the same value 1.
3. Initially, these object references are pointing to two distinct object locations in a heap space:



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?

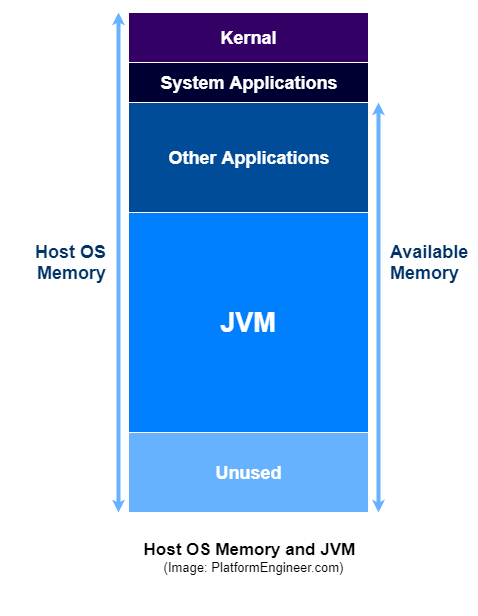
<https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973>

JVM Memory Model:

* You must have used some of the following JVM memory configurations when running resource-intensive Java programs.

|  |
| --- |
| -XmsSetting — initial Heap size  -XmxSetting — maximum Heap size  -XX:NewSizeSetting — new generation heap size  -XX:MaxNewSizeSetting — maximum New generation heap size  -XX:MaxPermGenSetting — maximum size of Permanent generation  -XX:SurvivorRatioSetting — new heap size ratios (e.g. if Young Gen size is  10m and memory switch is –XX:SurvivorRatio=2, then 5m will be  reserved for Eden space and 2.5m each for both Survivor spaces, default  value = 8)  -XX:NewRatio — providing ratio of Old/New Gen sizes (default value = 2) |

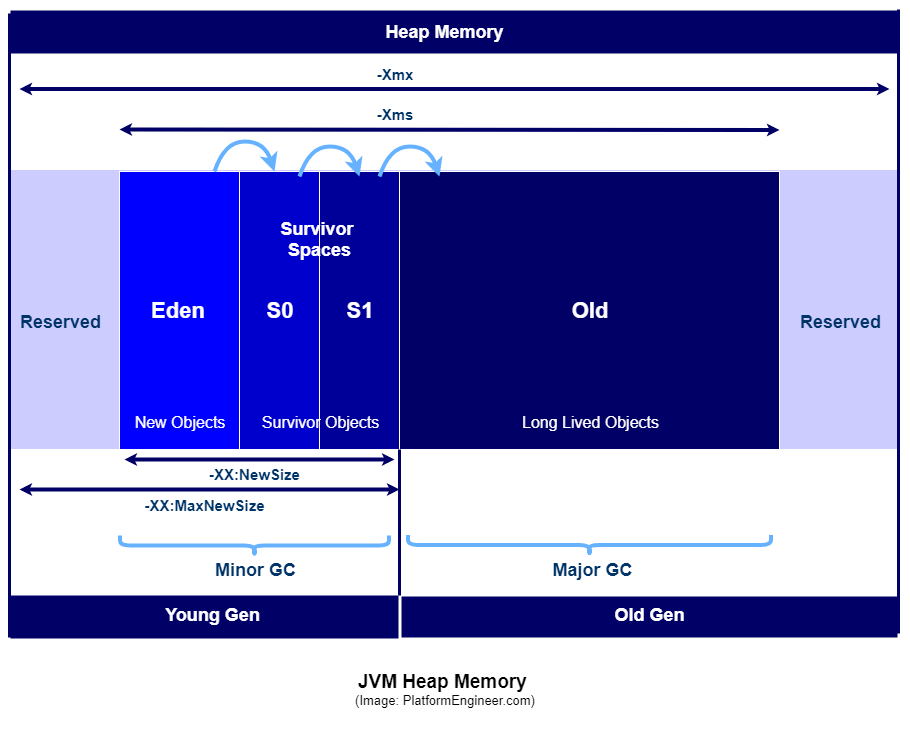
But have you ever wondered how your JVM resides on memory? Let me show it. Just like any other software, JVM consumes the available space on **host OS memory.**



However, inside JVM, there exist separate memory spaces (Heap, Non-Heap, Cache) in order to store runtime data and compiled code.

1 **Heap Memory:**

* Heap is divided into 2 parts — **Young Generation** and **Old Generation**
* Heap is allocated when JVM starts up (Initial size: -Xms)
* Heap size increases/decreases while the application is running
* Maximum size: -Xmx



## 1.1) Young Generation

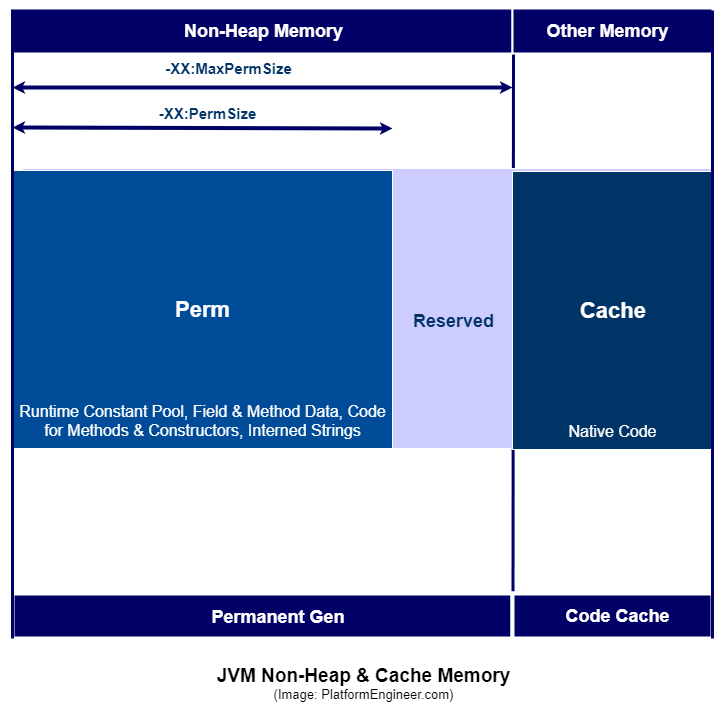
* This is reserved for containing newly-allocated objects
* Young Gen includes three parts — **Eden Memory and two Survivor Memory spaces (S0, S1)**
* Most of the newly-created objects goes Eden space.
* When Eden space is filled with objects, Minor GC (a.k.a. Young Collection) is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

## 1.2) Old Generation

* This is reserved for **containing long lived objects** that could survive after many rounds of Minor GC
* When Old Gen space is full, Major GC (a.k.a. Old Collection) is performed (usually takes longer time)

## 2) Non-Heap Memory

* This includes Permanent Generation (Replaced by Metaspace since Java 8)
* **Perm Gen stores per-class structures such** as runtime constant pool, **field and method data**, and the code for methods and constructors, as well as **interned Strings**
* Its size can be changed using **-XX:PermSize and -XX:MaxPermSize**

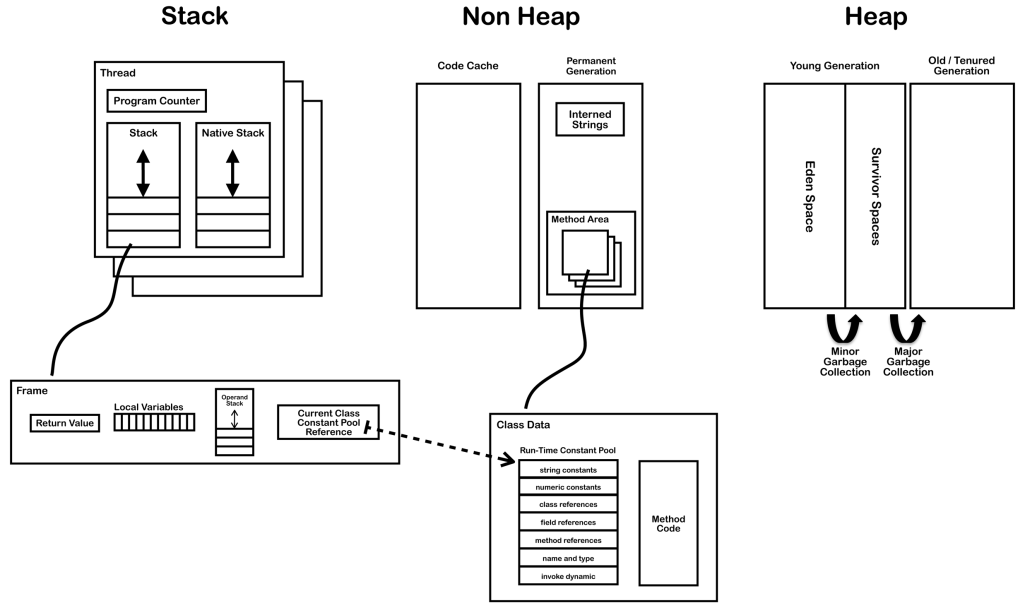


**3) Cache Memory**

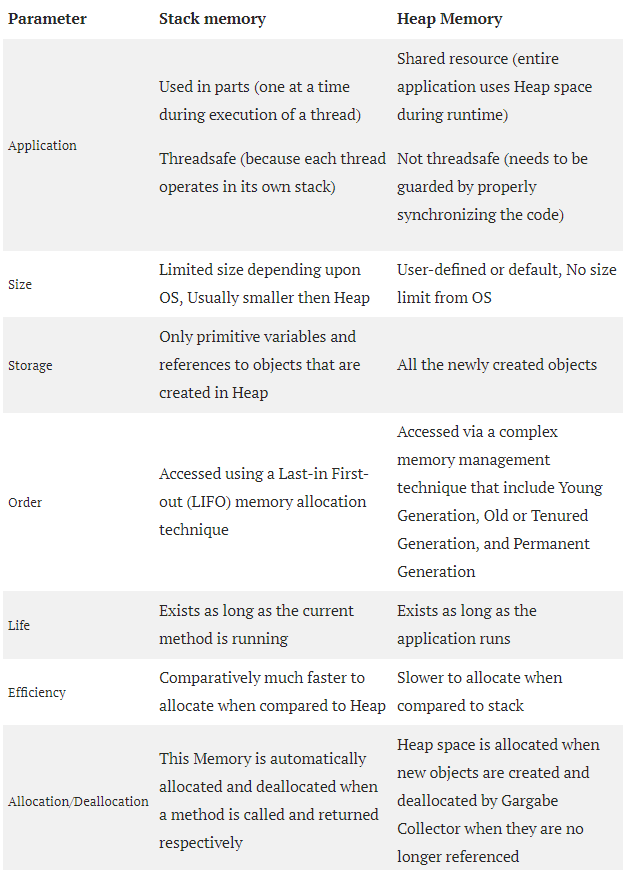
* This includes Code Cache
* **Stores compiled code (i.e. native code) generated by JIT compiler**, **JVM internal structures, loaded profiler agent code and data**, etc.
* When Code Cache exceeds a threshold, it gets flushed (and objects are not relocated by the GC).

**Stack vs. Heap**

So far I did not mention anything about Java Stack memory because I wanted to highlight its difference separately. First, take a look at the below image and check whether you know what’s happening here. I have already discussed on JVM Stack in my previous post. <https://platformengineer.com/2018/08/18/java-ecosystem-jvm-architecture/>

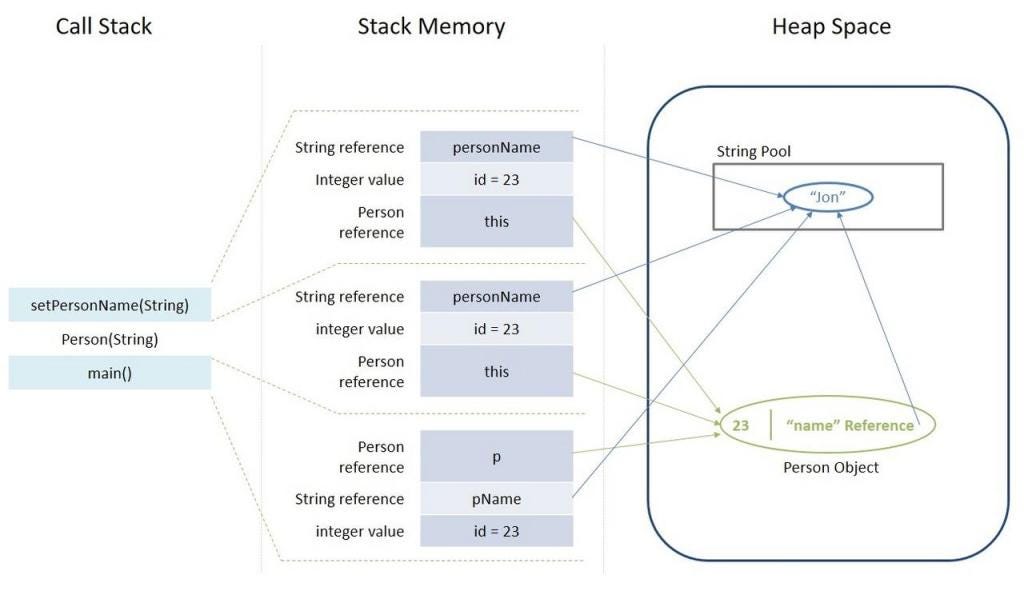


Java Stack memory is used for execution of a thread and it contains method specific values and references to other objects in Heap. Let’s put both Stack and Heap into a table and see their differences.



Here’s a nice example (from baeldung.com) on how Stack and Heap contribute to execute a simple program (Check the stack order with the code).

|  |
| --- |
| class Person {  int pid;  String name;  // constructor, setters/getters  }  public class Driver {  public static void main(String[] args) {  int id = 23;  String pName = "Jon";  Person p = null;  p = new Person(id, pName);  }  } |



**Latest JVM versions have different modifications such as introducing the following new memory spaces:**

The above Java memory model is the most commonly-discussed implementation. However, the latest JVM versions have different modifications such as introducing the following new memory spaces.

**Keep Area** — a new memory space in the Young Generation to contain the most recently allocated objects. No GC is performed until the next Young Generation. This area prevents objects from being promoted just because they were allocated right before a young collection is started.

**Metaspace** — Since Java 8, Permanent Generation is replaced by Metaspace. It can auto increase its size (up to what the underlying OS provides) even though Perm Gen always has a fixed maximum size. As long as the classloader is alive, the metadata remains alive in the Metaspace and can’t be freed.

**Memory Related Issues**

When there is a critical memory issue, the JVM gets crashed and throws an error indication in your program output like below.

1. java.lang.StackOverFlowError — indicates that Stack Memory is full
2. java.lang.OutOfMemoryError: Java heap space — indicates that Heap Memory is full
3. java.lang.OutOfMemoryError: GC Overhead limit exceeded — indicates that GC has reached its overhead limit
4. java.lang.OutOfMemoryError: Permgen space — indicates that Permanent Generation space is full
5. java.lang.OutOfMemoryError: Metaspace — indicates that Metaspace is full (since Java 8)
6. java.lang.OutOfMemoryError: Unable to create new native thread — indicates that JVM native code can no longer create a new native thread from the underlying operating system because so many threads have been already created and they consume all the available memory for the JVM
7. java.lang.OutOfMemoryError: request size bytes for reason — indicates that swap memory space is fully consumed by application
8. java.lang.OutOfMemoryError: Requested array size exceeds VM limit– indicates that our application uses an array size more than the allowed size for the underlying platform

**Metaspace vs PermGen space**

**PermGen (Permanent Generation)**

* PermGen (Permanent Generation) is **a special heap space separated from the main memory heap.**
* The JVM keeps track of loaded class metadata in the PermGen. Additionally, the JVM stores all the static content in this memory section. This includes all the static methods, primitive variables, and references to the static objects.  
  Furthermore, it contains data about bytecode, names, and JIT information. Before Java 7, the String Pool was also part of this memory. The disadvantages of the fixed pool size are listed in our write-up.

The default maximum memory size for 32-bit JVM is 64 MB and 82 MB for the 64-bit version.

However, we can change the default size with the JVM options:

|  |
| --- |
| -XX:PermSize=[size] is the initial or minimum size of the PermGen space  -XX:MaxPermSize=[size] is the maximum size |

Most importantly, Oracle completely removed this memory space in the JDK 8 release. Therefore, **if we use these tuning flags in Java 8 and newer versions,** we’ll get the following warnings:

|  |
| --- |
| >> java -XX:PermSize=100m -XX:MaxPermSize=200m -version  OpenJDK 64-Bit Server VM warning: Ignoring option PermSize; support was removed in 8.0  OpenJDK 64-Bit Server VM warning: Ignoring option MaxPermSize; support was removed in 8.0 |

* With its limited memory size, PermGen is involved in generating the famous **OutOfMemoryError**. Simply put, the class loaders weren’t garbage collected properly and, as a result, generated a memory leak.Therefore, we receive a memory space error; this happens mostly in the development environment while **creating** **new class loaders.**

**Metaspace**

* Simply put, Metaspace is a new memory space – starting from the Java 8 version; it has replaced the older PermGen memory space.
* The most significant difference is how it handles memory allocation.Specifically, this native memory region grows automatically by default.

We also have new flags to tune the memory:

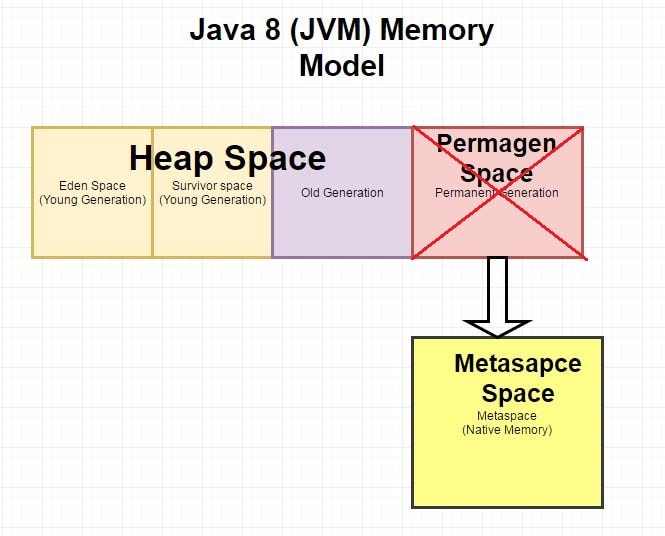
* MetaspaceSize and MaxMetaspaceSize – we can set the Metaspace upper bounds.
* MinMetaspaceFreeRatio – is the minimum percentage of class metadata capacity free after garbage collection
* MaxMetaspaceFreeRatio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space

**Benefits by replacing perm gem with MataSpace(native memory) :**

* Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers the cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size.
* Therefore, with this improvement, JVM reduces the chance to get the **OutOfMemory error.**
* Despite all of these improvements, we still need to monitor and tune the **metaspace to avoid memory leaks.**

**Notes:**

* q) Is permanent generation or meta-space is part of heap memory in java ?   
   a) From Java 8 PermGen was replaced by a new memory area called **Metaspace**. It has the same role as PermGen which is to store class metadata. **Metaspace is not part of the JVM heap, but is allocated in native operating system memory.**
* Metaspace, formerly known as PermGen (Permanent Generation), is a non-heap memory area that stores class metadata, constant pool information, and method bytecode.



**So** what is **Metaspace and how is it different from PermgenSpace.**

With JDK8, the permGen Space has been removed. So where will the metadata information be stored now? This metadata is now stored in a native memory are called as "MetaSpace". This memory is not a contiguous Java Heap memory. **It allows for improvements over PermGen space in Garbage collection, auto tuning, concurrent de-allocation of metadata.**

Difference between PermGen space and MetaSpace.

|  |  |
| --- | --- |
| PermGen Space | MetaSpace |
| PermGen always has a fixed maximum size. | Metaspace by default auto increases its size depending on the underlying OS. |
| Contiguous Java Heap Memory | Native Memory(provided by underlying OS) |
| Max size can be set using XX:MaxPermSize | Max size can be set using XX:MetaspaceSize |
| Comparatively inefficient Garbage collection. Frequent GC pauses and no concurrent deallocation. | Comparatively efficient Garbage collection. Deallocate class data concurrently and not during GC pause. |

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>

You know, sometimes we need to create Immutable classes for some purposes such as not allowing outside world to modify the objects. We have several inbuilt classes in Java also. Some of them are: String, Wrapper Classes, Arrays, LocalDate, LocalTime and etc...

**What is Immutability?**

* Immutability is the ability keep not changing with the modifications. Once it s defined no one can alter it
* Immutable class is a class which is once created, **its contents cannot be changed**
* You may remember String class! Right? It’s the most popular Immutable class in Java.
* So, how do we define a completely immutable custom class using Java? That’s the discussion I’ going to do…There are main steps to follow. I will first list them and later go into details.

**Steps:**

1. Declare class as final.
2. Make all properties as private final.
3. Do not declare setters. Only getters.
4. Declare all args constructor.
5. If there are custom nested objects in the class as properties, implement clone.
6. If there are other types of nested objects as properties, perform a deep copy.

**Step 1: Make all final**

We have to go for a final class because any class cannot override this class later. And all the properties must be private — then outside the class, they are not visible: encapsulated. When we make all properties final, they must be assigned with values within the constructor! Following these we will have the class like this.

|  |
| --- |
| **final** class Employee {  **private final String** empName;  private final int age;  private final Address address;  private final List<String> phoneNumbers;  private final Map<String, String> metadata;  public Employee(String name, int age, Address address, List<String> phoneNumbers, Map<String, String> metadata) {  super();  this.empName = name;  this.age = age;  this.address = address;  this.phoneNumbers = phoneNumbers;  this.metadata = metadata;  }  public String getEmpName() {  return empName;  }  public int getAge() {  return age;  }  public Address getAddress() {  return address;  }  public List<String> getPhoneNumbers() {  return phoneNumbers;  }  public Map<String, String> getMetadata() {  return metadata;  }  } |

Address:

|  |
| --- |
| final class Address {  private String street;  private String city;  public Address(String street, String city) {  this.street = street;  this.city = city;  }  public String getStreet() {  return street;  }  public String getCity() {  return city;  }  @Override  public String toString() {  return "{Street: " + street + ", City: " + city + "}";  }  } |

All good..Do you think so? Think what happens to phone numbers list and metadata map after we declare the object!! Anyone can add elements to both of them..But how?

|  |
| --- |
| Address address = new Address("street 1", "city X");  List<String> phoneNumbers = new ArrayList<>();  phoneNumbers.add("123456");  phoneNumbers.add("456789");  Map<String, String> metadata = new HashMap<>();  metadata.put("hobby", "Watching Movies");  // Declare the employee  Employee e = new Employee("John", 23, address, phoneNumbers, metadata);  // Update details  e.getPhoneNumbers().add("345123");  e.getMetadata().put("skill", "Java");  e.getMetadata().put("designation", "HR");  System.out.println(e.getPhoneNumbers());  System.out.println(e.getMetadata()); |

**Result**: Newly updated data have been highlighted here.

[123456, 456789, 345123]

{skill=Java, designation=HR, hobby=Watching Movies}

**See the phone numbers and metadata have been update! This breaks immutability! 😕**

We can even delete records from our “Immutable class”… You will loose data after this…

|  |
| --- |
| e.getMetadata().remove("hobby");  e.getPhoneNumbers().remove("123456"); |

This **breaks immutability again**! 😕 So, how we should manage this? See the next step..

**Step 2: Copy the objects and return**

* Next thing we have to do when we have nested Objects as properties is: take a copy of the existing data assigned using constructor and return as a new object.
* Then whenever you call GETTERS, you will get the old data. Not the updated ones.
* Let’s do this to getters of phoneNumbers and metadata.

|  |
| --- |
| final class Employee {  private final String empName;  private final int age;  private final Address address;  private final List<String> phoneNumbers;  private final Map<String, String> metadata;  public Employee(String name, int age, Address address, List<String> phoneNumbers, Map<String, String> metadata) {  super();  this.empName = name;  this.age = age;  this.address = address;  this.phoneNumbers = phoneNumbers;  this.metadata = metadata;  }  public String getEmpName() {  return empName;  }  public int getAge() {  return age;  }  public Address getAddress() {  return address;  }  // copy the list of phone numbers  public List<String> getPhoneNumbers() {  return new ArrayList<>(phoneNumbers);  }  // copy the map of metadata  public Map<String, String> getMetadata() {  return new HashMap<>(metadata);  }  } |

See..Even if we now add elements from client code it won’t e stored inside employee object**! It will always return the initialized objects**…Not we cannot add data into phoneNumbers and metadata. Cool 😎

Anything remaining❓

What about **Address** custom object? Let’s update it with some setters and see..

|  |
| --- |
| final class Address {  private String street;  private String city;  public Address(String street, String city) {  this.street = street;  this.city = city;  }  public String getStreet() {  return street;  }  public void setStreet(String street) {  this.street = street;  }  public String getCity() {  return city;  }  public void setCity(String city) {  this.city = city;  }  @Override  public String toString() {  return "{Street: " + street + ", City: " + city + "}";  }  } |

Can we update city and street in the **address** after declaring it in the client code?? YES!!! we can! See this.

|  |
| --- |
| e.getAddress().setCity("c3");  e.getAddress().setStreet("s3"); |

See the result..You will have the updated address inside Employee! It will be like this now => {Street: s3, City: c3}

This breaks immutability again! Still our class is **partially mutable** 😕 So, how we should manage this? See the next step..

**Step 3: Clone custom objects**

In a situation like this, we have to implement clone inside custom object: Address. Then Java will make sure that it is always giving a clone of the already declared object of address.

|  |
| --- |
| final class Address implements Cloneable {  private String street;  private String city;  public Address(String street, String city) {  this.street = street;  this.city = city;  }  public String getStreet() {  return street;  }  public void setStreet(String street) {  this.street = street;  }  public String getCity() {  return city;  }  public void setCity(String city) {  this.city = city;  }  public Object clone() throws CloneNotSupportedException {  return super.clone();  }  @Override  public String toString() {  return "{Street: " + street + ", City: " + city + "}";  }  } |

**Note**: And then when we get Address in Employee class, we should return a clone!

|  |
| --- |
| final class Employee {  private final String empName;  private final int age;  private final Address address;  private final List<String> phoneNumbers;  private final Map<String, String> metadata;  public Employee(String name, int age, Address address, List<String> phoneNumbers, Map<String, String> metadata) {  super();  this.empName = name;  this.age = age;  this.address = address;  this.phoneNumbers = phoneNumbers;  this.metadata = metadata;  }  public String getEmpName() {  return empName;  }  public int getAge() {  return age;  }    // clone the address object  public Address getAddress() throws CloneNotSupportedException {  return (Address) address.clone();  }  // copy the list of phone numbers  public List<String> getPhoneNumbers() {  return new ArrayList<>(phoneNumbers);  }  // copy the map of metadata  public Map<String, String> getMetadata() {  return new HashMap<>(metadata);  }  } |

Now try with the below client code and see the results...

|  |
| --- |
| Address address = new Address("street 1", "city X"); //address object  List<String> phoneNumbers = new ArrayList<>();  phoneNumbers.add("123456");  phoneNumbers.add("456789");  Map<String, String> metadata = new HashMap<>();  metadata.put("hobby", "Watching Movies");  // Declare the employee  Employee e = new Employee("John", 23, address, phoneNumbers, metadata);  //e should be immutable, if any body try to change any of the properties of that object then it should not allow to modify.  // Update details  e.getPhoneNumbers().add("345123");//get() will give new arraylist always and your going to add elements on new collection object not origin “e” object  e.getMetadata().put("skill", "Java"); //get() will return new map always and your going to insert bucket in new Map object  e.getMetadata().put("designation", "HR");//get() will return new map always and your going to insert bucket in new Map object    // change address details  e.getAddress().setCity("c3");//It will set the object always in new object and not in origin la address object when you create  //Note: e.getAddress() – it will give cloned object from Object class, not original object from “e” related address.  e.getAddress().setStreet("s3");  System.out.println(e.getPhoneNumbers());  System.out.println(e.getMetadata());  System.out.println(e.getAddress()); |

Is the Employee object changed? NO..Right? Now it’s keeping constant against whatever the change we do to change its data!

Now we have achieved IMMUTABILITY 😍 💪 😊

|  |
| --- |
| Final class setup will be like this with client code.  import java.util.ArrayList;  import java.util.HashMap;  import java.util.List;  import java.util.Map;  public class ImmutableClassDemo {    public static void main(String[] args) throws CloneNotSupportedException {    Address address1 = new Address("s1", "c1");  List<String> phoneNumbers = new ArrayList<>();  phoneNumbers.add("123345");  phoneNumbers.add("456789");  Map<String, String> metadata = new HashMap<>();  metadata.put("hobby", "Watching Movies");  Employee e = new Employee("John", 23, address1, phoneNumbers, metadata);  // modifications  e.getAddress().setCity("c3");  e.getAddress().setStreet("s3");  e.getPhoneNumbers().add("1234");  e.getMetadata().put("skill", "Java");  e.getMetadata().put("designation", "HR");  System.out.println(e.getEmpName());  System.out.println(e.getAge());  System.out.println(e.getAddress());  System.out.println(e.getPhoneNumbers());  System.out.println(e.getMetadata());  }  }  final class Employee {  private final String empName;  private final int age;  private final Address address;  private final List<String> phoneNumbers;  private final Map<String, String> metadata;  public Employee(String name, int age, Address address, List<String> phoneNumbers, Map<String, String> metadata) {  super();  this.empName = name;  this.age = age;  this.address = address;  this.phoneNumbers = phoneNumbers;  this.metadata = metadata;  }  public String getEmpName() {  return empName;  }  public int getAge() {  return age;  }  // clone the address object  public Address getAddress() throws CloneNotSupportedException {  return (Address) address.clone();  }  // deep copy the list of phone numbers  public List<String> getPhoneNumbers() {  return new ArrayList<>(phoneNumbers);  }  // deep copy the map of metadata  public Map<String, String> getMetadata() {  return new HashMap<>(metadata);  }  }  final class Address implements **Cloneable** {  private String street;  private String city;  public Address(String street, String city) {  this.street = street;  this.city = city;  }  public String getStreet() {  return street;  }  public void setStreet(String street) {  this.street = street;  }  public String getCity() {  return city;  }  public void setCity(String city) {  this.city = city;  }  public Object clone() throws CloneNotSupportedException {  return super.clone();  }  @Override  public String toString() {  return "{Street: " + street + ", City: " + city + "}";  }  } |

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.

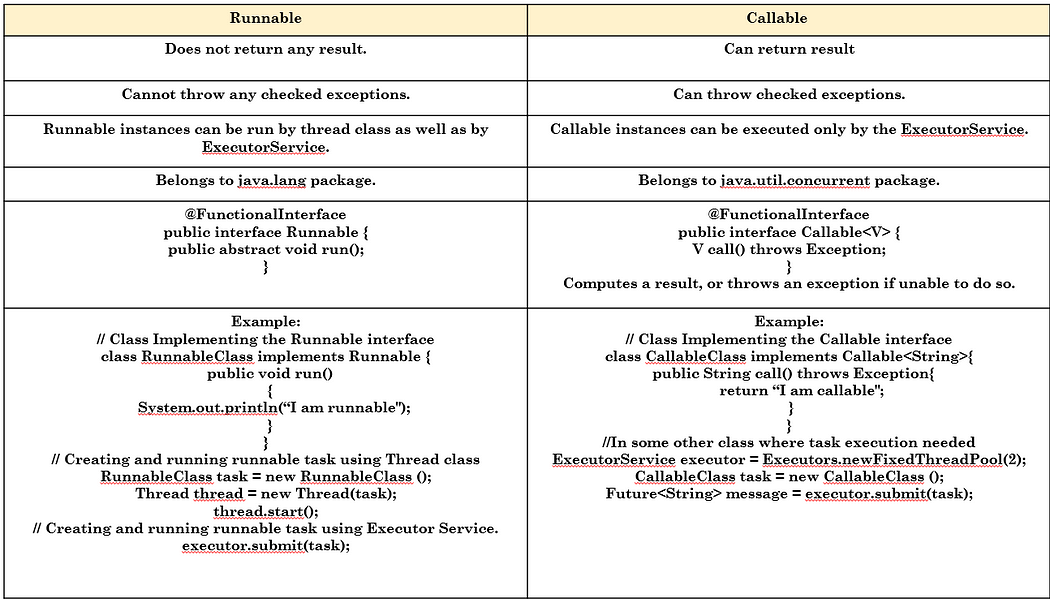
**What is Future?**

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. |

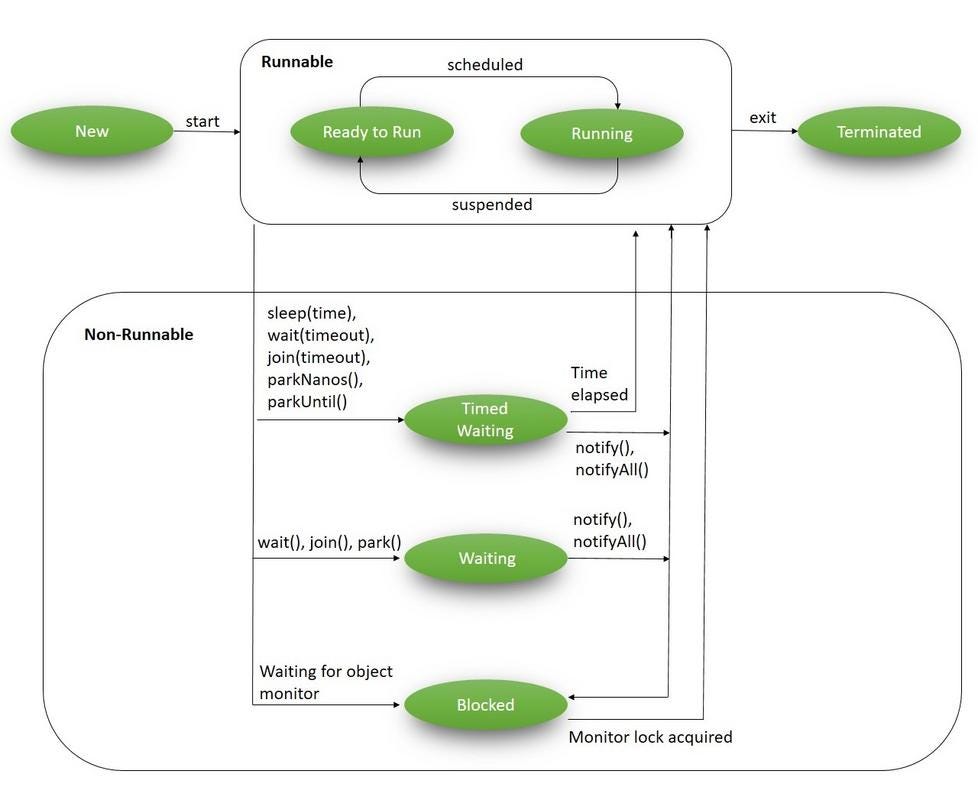
q) Can a class implement both Callable and Runnable interfaces?

**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:**

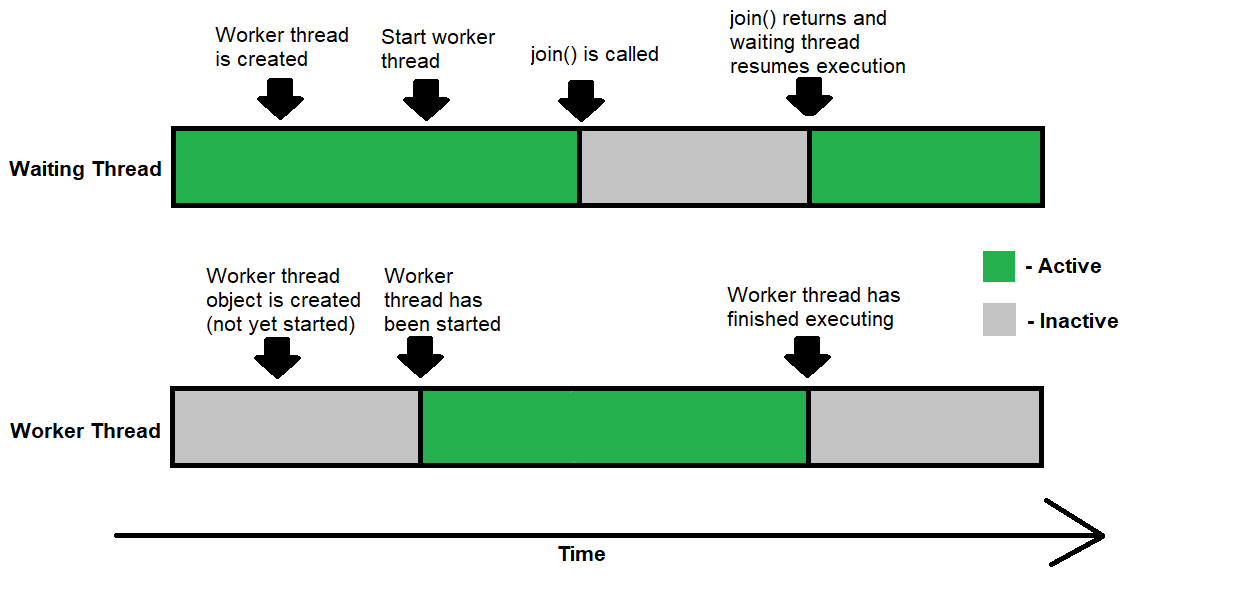
* When we interrupt running thread by calling/invoking any of the following method then it will go to Non-Runnable state.
* **Join(), wait(), yield(), notify(), notifyAll(), sleep() and park()** etc.
* **Blocked/Waiting state**: When a thread is temporarily inactive, then it’s in one of the following 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 |

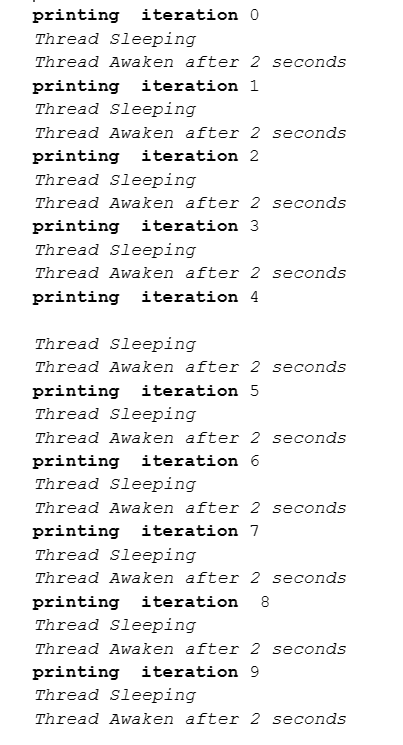
1. **public static void sleep(long millis) throws InterruptedException | public static void sleep(long millis, int nanos) throws InterruptedException | public static void sleep(Duration duration) throws InterruptedException**

**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

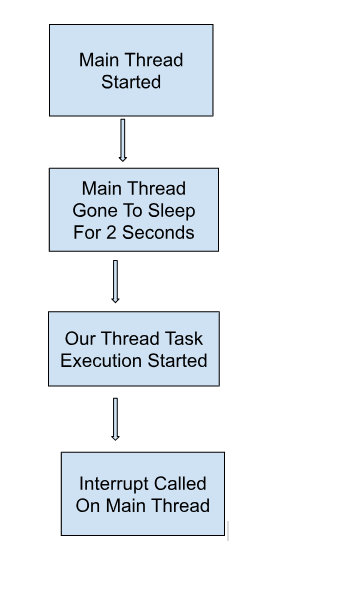
|  |
| --- |
| public class InterruptHandler implements Runnable {  @Override  public void run() {  try {  System.out.println(Thread.currentThread().getName() + " : " + "Task Started");  System.out.println(Thread.currentThread().getName() + " : "+ "Sleeping the thread for 5 seconds");  Thread.sleep(5000);  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.

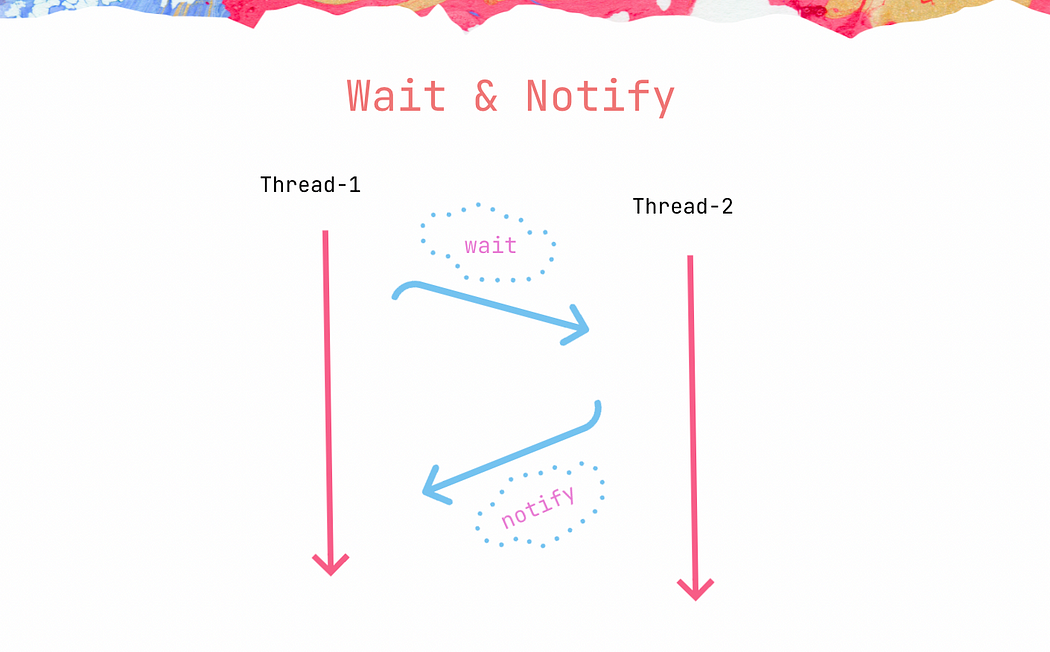
**Explanation:**

* 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.

**wait() && notify() && notifyAll()Method:**

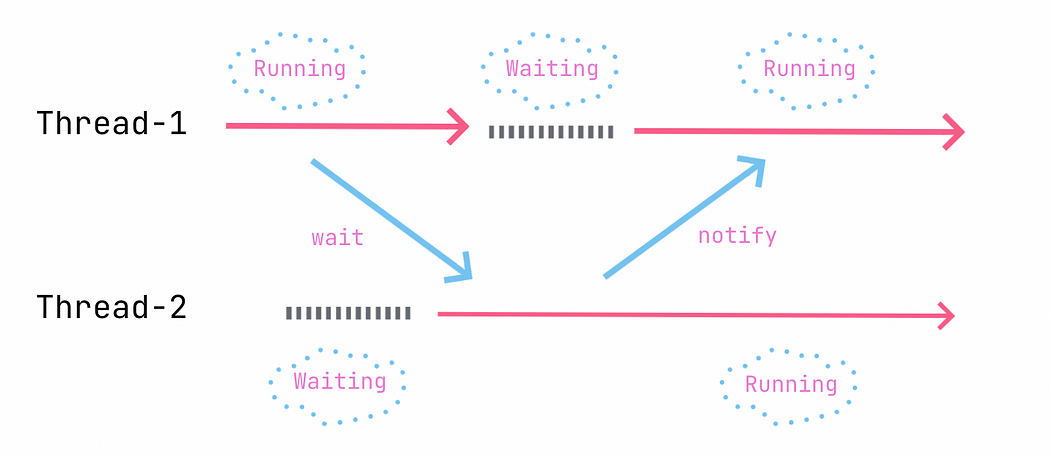
1. **public final void wait(); throws InterruptedException || public final void wait(long timeoutMillis) throws InterruptedException || public final void wait(long timeoutMillis, int nanos) throws InterruptedException**

Reference link: <https://java-jedi.medium.com/multithreading-wait-notify-3002ecd0b555>



* These methods are mainly used to enable inter-communication of threads during their execution. They are used in situations when one thread may release lock on some common shared resource for some time, so that another thread may access shared resource, execute and relinquish the lock to the first thread. Then, the first thread continues its execution.
* Since all the threads depend on intrinsic lock of a given object, we can implement inter-thread communication. It is possible with the help **of wait() and notify().** Threads that are locking on the same intrinsic lock, or monitor, can release lock until the other thread calls notify.

|  |
| --- |
| **wait ()** — owner thread releases the lock, the other thread waiting for the lock can acquire the lock and start execution. Owner thread will wait until it is notified or interrupted. If thread is interrupted while wait(), it throws InterruptedException  **notify()** — as soon as the thread finishes execution, it calls notify() to return the lock to the owner |



Let’s discuss the illustration above. As soon as thread#1 calls wait(), thread#2 starts executing. When thread#2 calls notify() -> the lock is returned back to thread#1 and it will continue executing.

|  |
| --- |
| **Note**: these methods wait() and notify() are used and called from synchronized blocks or methods since at the time of calling these methods, thread must be the owner of the lock, otherwise **IllegalMonitorStateException** will be thrown. |

|  |
| --- |
| **Let’s consider the following example.**  public void produce() throws InterruptedException {  synchronized (this) {  System.out.println("Initial value: " + value);  value++;  System.out.println("Producer is executing, incrementing value\n Value = " + value);  wait();  System.out.println("-------------------------");  System.out.println("""  Continuing executing after passing the lock to another thread. The lock has been acquired  """);  System.out.println("Value: " + value);  }  }  public void consume() throws InterruptedException {  Thread.sleep(1000);  synchronized (this) {  System.out.println("-------------------------");  System.out.println("Consumer started executing");  System.out.println("Value: " + value);  value++;  System.out.println("Value is incremented, value = " + value);  notify();  System.out.println("Notifying the other thread and releasing the lock");  }  }  }    public class App {  public static void main(String[] args) {  Processor processor = new Processor();  Thread threadOne = new Thread(() -> {  try {  processor.produce();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  Thread threadTwo = new Thread(() -> {  try {  processor.consume();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  threadOne.start();  threadTwo.start();  }  }  **Output:**  // Output  Initial value: 1 //thread#1  Producer is executing, incrementing value //thread#1  Value = 2 //thread#1  - - - - - - - - - - - - -  Consumer started executing //thread#2  Value: 2 //thread#2  Value is incremented, value = 3 //thread#2  Notifying the other thread and releasing the lock //thread#2  - - - - - - - - - - - - -  Continuing executing after passing the lock to another thread. The lock has been acquired. //thread#1  Value: 3 //thread#1 |

**Explanation:**

It is a pretty straightforward example. There are two threads — **thread#1, thread#2** and processor variable of class Processor. This Processor class has two synchronized methods — produce() and consume(). As the program runs, thread#1 calls produce() method and thread#2 calls consumer()

First, thread#1 acquires the lock, does some operations and calls wait(), at the same time, thread#2 already started execution and tries to acquire the lock that is owned by thread#1. As soon as thread#1 calls wait(), it enters a blocked / waiting state and releases the lock and thread#2 acquires the lock and continues execution; thread#2 does some operations and then calls notify() to return the lock to thread#1. Thread#1 then acquires the lock and resumes execution. This is a simple approach to achieve inter-thread communication.

**Caveats of using wait() and notify()**

Improper usage of wait() and notify() causes resource leakage that may lead to crashing of your system. Let’s discuss the following example.

|  |
| --- |
| class Processor {  int value = 1;  public void produce() throws InterruptedException {  synchronized (this) {  System.out.println("Initial value: " + value);  value++;  System.out.println("Producer is executing, incrementing value\nValue = " + value);  System.out.println("-------------------------");  wait();  System.out.printf("""  Continuing executing after passing the lock to another thread. The lock has been acquired.  Value: %s  """, value);  }  }  public void consume() throws InterruptedException {  Thread.sleep(1000);  synchronized (this) {  System.out.println("-------------------------");  System.out.println("Consumer started executing");  System.out.println("Value: " + value);  value++;  }  }  }  public class App {  public static void main(String[] args) throws InterruptedException {  Processor processor = new Processor();  Thread threadOne = new Thread(() -> {  try {  processor.produce();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  Thread threadTwo = new Thread(() -> {  try {  processor.consume();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  threadOne.start();  threadTwo.start();  }  } |

**Explanation:**

This example is nearly identical to the example above with a small difference — there is no line where notify() is called in method consume(). The program flow is the same — thread#1 starts execution, acquires the lock, does some operation and then calls wait(). In the meantime, thread#2 acquires the lock after thread#1 call wait(), does some operations and finishes its execution. Thread#2 never calls notify() and thread#1 never acquires the lock again and remains blocked / waiting forever. This causes to resource leakage, the thread never finishes execution and cannot be used again or freed up, ultimately eating up resources and leading to crashing of the application overtime. Similar situation occurs when a thread dies before calling notify(). Hence, it is very important to call notify() to avoid this problem.

Additionally, to avoid this problem, it is possible to use other methods provided — wait(int timeoutMillis) and wait(int timeoutMillis, int nanos).

The thread will wait for specified amount of time and resumes execution if not notified.

**notify() & notifyAll()**

* There is another method — notifyAll() to relinquish the lock. The difference between them is that notify() wake up single thread and notifyAll() wake all the threads competing to acquire the lock.
* Inter-thread communication is commonly used in Pub/Sub pattern, where one thread acts a producer and the other acts as a consumer. To implement such systems, you need a proper synchronization mechanism and enable inter-thread communication between threads.
* In this series of posts in Multithreading, we have discussed all the necessary concepts related mythical methods — wait() and notify(), how to implement inter-thread communication and caveats of it. Next, we will discuss locking mechanisms — Lock and its implementation ReentrantLock. Subscribe and stay tuned to be the first to read new posts.

**Can we use wait and notify without synchronized methods?**

Reference link:[**https://www.oreilly.com/library/view/java-threads-second/1565924185/ch04s02.html#:~:text=Actually%2C%20the%20answer%20is%20no,wait%20and%20notify%20mechanism%20itself**](https://www.oreilly.com/library/view/java-threads-second/1565924185/ch04s02.html#:~:text=Actually%2C%20the%20answer%20is%20no,wait%20and%20notify%20mechanism%20itself)**.**

* Wait and notify does not solve the race condition problem that the **synchronized mechanism solves.** As a matter of fact, wait and notify must be used in conjunction with the synchronized lock to prevent a **race condition** in the wait and notify mechanism itself.
* **Can wait and notify be used to replace the synchronized method?** Actually, the answer is no. Wait and notify does not solve the race condition problem that the synchronized mechanism solves. As a matter of fact, wait and notify must be used in conjunction with the synchronized lock to prevent a race condition in the wait and notify mechanism itself.

**General Differences between wait() and sleep()**

* Simply put, wait() is an instance method that’s used for thread synchronization.
* It can be called on any object, as it’s defined right on java.lang.Object, but it can only be called from a synchronized block. It releases the lock on the object so that another thread can jump in and acquire a lock.
* On the other hand, Thread.sleep() is a static method that can be called from any context. Thread.sleep() pauses the current thread and does not release any locks.

**Difference between join() and wait() method in java**

* One most important difference between wait() and join() that is wait() must be called from synchronized context i.e. synchronized block or method otherwise it will throw IllegalMonitorStateException but On the other hand, we can call join() method with and without synchronized context in Java.
* yield does not take or release locks, it simply pauses the current thread execution. So yielding in the synchronized block will not let the current thread to release lock and let the other methods to enter the synchronized block. wait/notify method should be used to release the lock
* Join method in Java allows one thread to wait until another thread completes its execution. In simpler words, it means it waits for the other thread to die. It has a void type and throws InterruptedException.

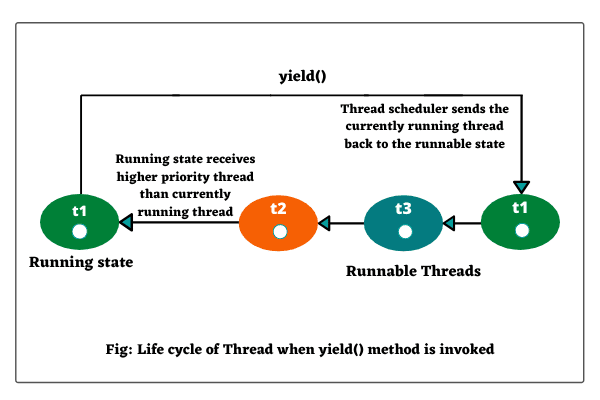
|  |
| --- |
| **Note**: In Java Multi-threading concepts, only wait() and notify() methods only will talk about lock/monitor concept i.e acquiring lock or releasing lock and other all methods(sleep(), join(), yield() methods) will talk about making current thread from running state to non-runnable state. |

**yield() method :** (private static native void yield0();)

A **yield()** method is a static method of Thread class and it can stop the currently executing thread and will give a chance to other waiting threads of the same priority. If in case there are no waiting threads or if all the waiting threads have low priority then the same thread will continue its execution.

Yield() method in Java | When a currently executing thread goes to the runnable state from running state, this process is called yield in Java.

Reference Link: <https://www.scientecheasy.com/2020/08/java-yield-method.html/>



|  |
| --- |
| Program based on yield() method in Java  Let’s create a Java program in which we will move back current executing thread to the runnable state from running state. Look at the following source code.  Program code:  public class A implements Runnable  {  public void run()  {  System.out.println(Thread.currentThread());  Thread.yield(); // Calling yield() method on current thread to move back into the runnable state from running state.  System.out.println(Thread.currentThread());  }  public static void main(String[] args)  {  A a1 = new A();  Thread t1 = new Thread(a1, "First Child Thread");  A a2 = new A();  Thread t2 = new Thread(a2, "Second Child Thread");  t1.start();  t2.start();  }  }  Output:  Thread[First Child Thread,5,main]  Thread[Second Child Thread,5,main]  Thread[First Child Thread,5,main]  Thread[Second Child Thread,5,main] |

**Explanation:**

1. As you can observe in the output, first t1 starts running. When yield() method is invoked, thread t1 goes to the runnable state from running state, and thread t2 starts execution. Then t2 goes into the runnable state because yield() method is called on it.

2. Now, t1 resumes its execution. After complete execution of t1, t1 exits and terminated. Then t2 resumes for the execution.

Let’s take another program in which we will set the priority and then we will call yield() method. Look at the source code.

|  |
| --- |
| Program code:  public class A implements Runnable  {  public void run()  {  System.out.println(Thread.currentThread());  Thread.yield(); // Calling yield() method on current thread to move back into the runnable state from running state.  System.out.println(Thread.currentThread());  }  public static void main(String[] args)  {  A a1 = new A();  Thread t1 = new Thread(a1, "First Child Thread");  A a2 = new A();  Thread t2 = new Thread(a2, "Second Child Thread");  t1.setPriority(4); // Setting the priority of thread  t2.setPriority(8);  t1.start();  t2.start();  }  }  Output:  Thread[Second Child Thread,8,main]  Thread[First Child Thread,4,main]  Thread[Second Child Thread,8,main]  Thread[First Child Thread,4,main] |

**Explanation:**

1. Priority of thread t1 is set to 4 whereas priority of thread t2 is set to 8.

2. Since thread t2 has the highest priority so thread scheduler selects it for execution. When yield() method is called on thread t2, t2 goes to the ready state.

3. Now scheduler selects thread t1 for its execution. When yield() method is invoked, t1 goes to the ready state.

4. Since thread t2 enjoys the highest priority, so scheduler again selects it for execution.

5. After the complete execution of t2, thread t2 terminated, and then t1 starts execution

Hope that this tutorial has covered important points related to yield() method in Java with example programs. Keep in mind the following important points.

**Key Points to Remember**

1. Do not use yield() method constantly in Java program.

2. When the yield() method is called on a thread object, it does not move the thread into sleeping, waiting, or blocking state. It sends thread into runnable state that can be resumed to the running state later.

**Executor Framework in Java:**

Reference link: [**https://medium.com/javarevisited/a-complete-guide-on-executorservice-in-java-67528f1a535b#:~:text=The%20Executor%20Framework%20is%20a,logic%20rather%20than%20thread%20management**](https://medium.com/javarevisited/a-complete-guide-on-executorservice-in-java-67528f1a535b#:~:text=The%20Executor%20Framework%20is%20a,logic%20rather%20than%20thread%20management)**.**

* 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(I), Executors(C-Utility factory methods), ExecutorService(I), ScheduledExecutorService(I), ScheduledThreadPoolExecutor (C) and ThreadPoolExecutor(C)**. 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.** |

* When an ExecutorService is terminated then it has no tasks actively executing, no tasks waiting for execution, and no new tasks can be submitted.
* Hence an unused ExecutorService should be shut down to allow the reclamation of its resources.

Let’s See Some Methods of ExecutorService:

* submit(): this method accepts a runnable or callable task and returns a Future that can be used to wait for completion and/or to cancel execution.
* invokeAny(): this method accepts a collection of callable tasks and returns a result if any tasks are successful.
* invokeAll(): this method accepts a collection of callable tasks and returns a List of Future, which will hold the result returned by each task when the asynchronous tasks are completed.

Example Program:

|  |
| --- |
| //Create executorService using ThreadPoolExecutor implementation of  //ExecutorService.  ExecutorService executorService = new ThreadPoolExecutor(1, 5, 0L,  TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>());  //Create callable tasks using lambda implementation of call method of  //Callable Interface  Callable<String> callableTask = () -> {  System.out.println("Call method called.");  TimeUnit.MILLISECONDS.sleep(2000);  return "Task execution in call method";  };  //submit single callable task to executorService, that is returning a Future  Future<String> future = executorService.submit(callableTask);  System.out.println(future.get());  //Create list of callable tasks  List<Callable<String>> callableTasks = new ArrayList<>();  callableTasks.add(callableTask);  callableTasks.add(callableTask);  callableTasks.add(callableTask);  //submit the list of callable tasks to invokeAny method that returns a result.  String result = executorService.invokeAny(callableTasks);  System.out.println(result);  //submit the list of callable tasks to invokeAll method that returns a list  // of futures representing results of asynchronous tasks.  List<Future<String>> futures = executorService.invokeAll(callableTasks);  //shutdown the executorService after completing all tasks to reclaim memory.  executorService.shutdown(); |

**What is ThreadPoolExecutor?**

* The ThreadPoolExecutor is an implementation of ExecutorService and provides a pool of threads that execute the runnable or callable tasks.
* There is also an Executors class that has a set of factory methods to create different types of thread pools.
* The below code that creates new ThreadPoolExecutor can be replaced by one such factory method of Executors class. But before that let’s understand ThreadPoolExecutor and its parameters.

|  |
| --- |
| ExecutorService executorService = new ThreadPoolExecutor(1, 5, 0L,  TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>()); |

The above example ThreadPoolExecutor creates a new thread pool executor with the given initial parameters.

Where Params Are as Below:

* corePoolSize — the number of threads to keep in the pool, even if they are idle unless allowCoreThreadTimeOut is set.
* maximumPoolSize — the maximum number of threads to allow in the pool.
* keepAliveTime — when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.
* unit — the time unit for the keepAliveTime argument.
* workQueue — the queue to use for holding tasks before they are executed. This queue will hold only the Runnable tasks submitted by the execute.

**But the above code can be replaced by a factory method of the Executors class:**

|  |
| --- |
| ExecutorService executorService = Executors.newFixedThreadPool(10); |

**The advantages of using Thread pools are that it addresses two different problems:**

* They usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead.
* The ThreadPoolExecutor provides an efficient way of managing threads and resources associated with it while executing a huge number of async tasks.

**shutdown() and shutdownNow():**

* The shutdown() method of ExecutorService will allow previously submitted tasks to execute before terminating the ExecutorService and performing memory clean-up, whereas the shutdownNow() method prevents waiting for tasks from starting and attempts to stop currently executing tasks.
* This shutdownNow() returns a list of tasks that are waiting to be processed. It is up to the developer to decide what to do with these tasks.

|  |
| --- |
| List<Runnable> notExecutedTasks = **executorService.shutDownNow();** |

**Note**:

* One good way to shut down the ExecutorService (which is also recommended by Oracle) is to use both of these methods combined with the awaitTermination() method.
* awaitTermination: Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
* This method returns true if this executor is terminated successfully and false if the timeout elapsed before the termination of this executor.

**Sample Code:**

|  |
| --- |
| executorService.shutdown();  try {  if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {  executorService.shutdownNow();  }  } catch (InterruptedException e) {  executorService.shutdownNow();  }"); |

**With this approach, the ExecutorService will first stop taking new tasks and then wait up to a specified period for all tasks to be completed. If that time expires, the execution is stopped immediately.**

**The Future Interface: (Future is not Functional Interface)**

|  |
| --- |
| **public interface Future<V> {**  **V get() throws InterruptedException, ExecutionException;** |

* A Future is the result of asynchronous tasks that may be completed in the **future**.
* Future also provides methods to check if the **computation is complete**, to wait for its **completion**, and to retrieve the result of the computation(example get() method).

|  |
| --- |
| What is meaning of computation generally? - గణన  1.: the act or action of computing : calculation. 2. : a system of calculating especially by mathematical means. |

**Examples:**

|  |
| --- |
| Future<String> future =  executorService.submit(callableTask);  System.out.println(future.get());  List<Future<String>> futures = executorService.invokeAll(callableTasks); |

Explanation:

* In the above code get() method is used to get the result of the async task, if (true) **completed** else **block until it becomes available**(**in this case main thread will be blocked until the result of the async task becomes available**).
* With very long blocking caused by the get() method, an application’s performance can degrade. If the resulting data is not crucial, it is possible to avoid such a problem by using timeouts:

|  |
| --- |
| String result = future.get(200, TimeUnit.MILLISECONDS); |

* If the execution period is longer than specified (in this case, 200 milliseconds), a TimeoutException will be thrown.
* We can use the **isDone()** method to check if the assigned task has already been processed or not.
* The Future interface also provides **for canceling task execution with the cancel() method** and checking the cancellation with **the isCancelled(**) method:

|  |
| --- |
| boolean canceled = future.cancel(true);  boolean isCancelled = future.isCancelled(); |

**Limitations of Future:**

* **It can not be manually completed**
* Let’s say that you’ve written a function to fetch the latest price of an e-commerce product from a remote API. Since, this API call is time consuming, you’re running it in a separate thread and returning a Future from your function.Now, let’s say that If the remote API service is down, then you want to complete the Future manually by the last cached price of the product.
* Can you do this with Future? No!
* You can not perform further action on a Future’s result without blocking Future does not notify you about its completion. It provides a get() method which blocks until the result is available.
* You don’t have the ability to attach a callback function to the Future and have it get called automatically when the Future’s result is available.
* Multiple Futures can not be chained together, sometimes you need to execute a long running computation and when the computation is done, you need to send its result to another long running computation, and so on.
* You can not create such asynchronous work flow with Futures.
* You can not combine multiple Futures together
* Let’s say that you have 10 different Futures that you want to run in parallel and then run some function after all of them completes. You can’t do this as well with Future.

**No Exception Handling**

* Future API does not have any exception handling construct.Whoa! So many limitations right? Well, That’s why we have CompletableFuture. You can achieve all of the above with CompletableFuture.
* CompletableFuture implements Future and CompletionStage interfaces and provides a huge set of convenience methods for creating, chaining and combining multiple Futures. It also has a very comprehensive exception handling support.

**ExecutorService(java 1.5) vs Fork/Join Framework(java 1.7):**

Reference Link: <https://jackynote.medium.com/how-i-can-understand-a-simple-about-executorservice-and-fork-join-framework-dead7fe0ed77>

* Java is a popular programming language that offers robust support for concurrency, making it an excellent choice for developing multi-threaded applications.
* Two key components in Java that simplify the process of managing and executing tasks concurrently are the ExecutorService and Fork/Join Framework.

ExecutorService:

ExecutorService is like the manager of a group of workers (threads) that helps you execute tasks concurrently in a controlled manner. It abstracts away much of the complexity involved in managing threads. Let’s dive into a simple example to illustrate its usage.

Example:

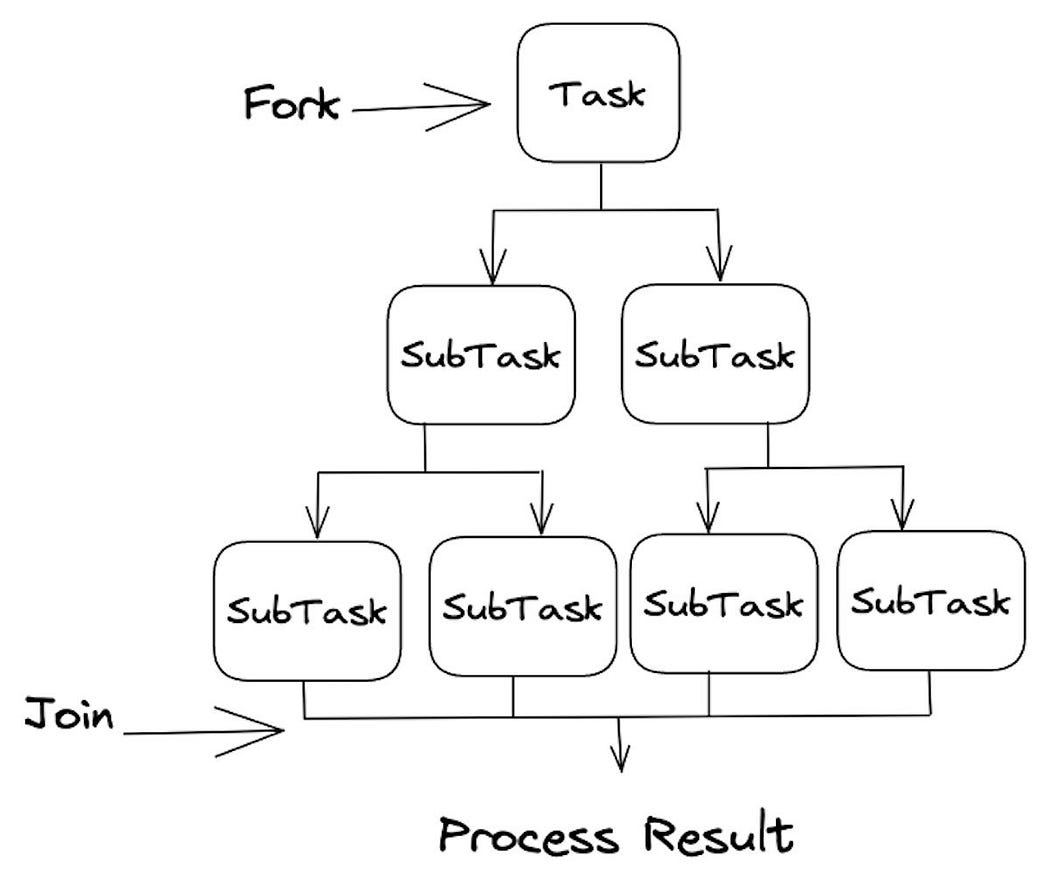
|  |
| --- |
| import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  public class ExecutorServiceExample {  public static void main(String[] args) {  // Create an ExecutorService with a fixed pool of 4 threads  ExecutorService executorService = Executors.newFixedThreadPool(4);  // Submit tasks for execution  for (int i = 0; i < 10; i++) {  final int taskNumber = i;  executorService.execute(() -> {  System.out.println("Task " + taskNumber + " is running on thread " + Thread.currentThread().getName());  });  }  // Shutdown the ExecutorService when done  executorService.shutdown();  }  } |

Explanantion:

In this example, we:

* Create an ExecutorService with a fixed pool of 4 threads using Executors.newFixedThreadPool(4).
* Submit 10 tasks for execution using the execute method.
* The ExecutorService manages the allocation and reuse of threads for these tasks.
* We shut down the ExecutorService once we're done with it.

Fork/Join Framework:



* The Fork/Join Framework is a more specialized tool designed for parallel processing. It is ideal for situations where you need to divide a large task into smaller sub-tasks and then combine their results.
* Let’s understand it with a classic example — calculating the sum of elements in an array.

Example:

|  |
| --- |
| import java.util.concurrent.RecursiveTask;  import java.util.concurrent.ForkJoinPool;  public class ForkJoinExample {  public static void main(String[] args) {  int[] data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  ForkJoinPool pool = new ForkJoinPool();  int result = pool.invoke(new SumTask(data, 0, data.length));  System.out.println("Sum: " + result);  }  }  class SumTask extends RecursiveTask<Integer> {  private int[] data;  private int start;  private int end;  public SumTask(int[] data, int start, int end) {  this.data = data;  this.start = start;  this.end = end;  }  @Override  protected Integer compute() {  if (end - start <= 3) { // Small enough, do the task directly  int sum = 0;  for (int i = start; i < end; i++) {  sum += data[i];  }  return sum;  } else {  int mid = start + (end - start) / 2;  SumTask leftTask = new SumTask(data, start, mid);  SumTask rightTask = new SumTask(data, mid, end);  leftTask.fork(); // Fork new tasks-- the leftTask is allowed to execute in parallel.  int rightResult = rightTask.compute(); // Compute one part  int leftResult = leftTask.join(); // Wait for the other part  return leftResult + rightResult;  }  }  } |

Explanation:

In this example, we:

* Initialize an array of data.
* Create a ForkJoinPool, which is the **heart of the Fork/Join Framework**.
* Define a SumTask class that extends RecursiveTask<Integer>. This is where we specify the task to be performed.
* The SumTask class breaks down the task of summing the array into **smaller sub-tasks and combines the results.** The framework handles task distribution and result **aggregation** for us.

**How the Fork/Join Framework Works**

The Fork/Join Framework employs a **divide-and-conquer strategy** to parallelize tasks. Here’s a step-by-step breakdown of how it works:

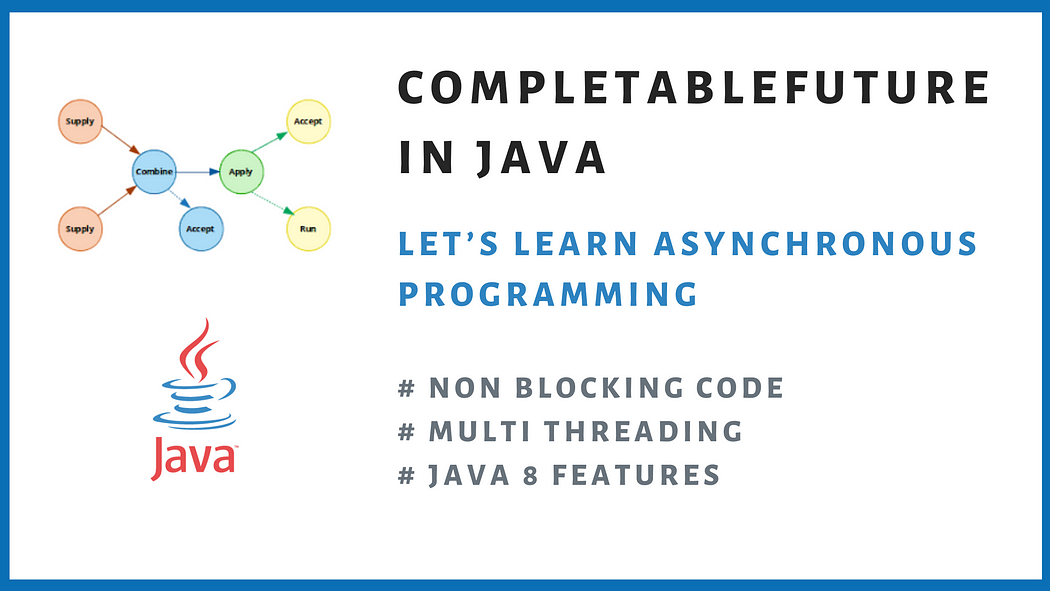
1. The main task, in this case, the SumTask, is given a large problem to solve. It checks if the problem size is small enough to be solved directly (in this case, when the difference between start and end is less than or equal to 3).
2. If the problem size is small, it computes the result directly. In this example, it sums the elements in the array between start and end.
3. If the problem is too large, the main task splits it into two sub-tasks, leftTask and rightTask, to solve smaller parts of the problem.
4. The leftTask is then forked, which means it's submitted for parallel execution. This allows the framework to allocate separate threads for each sub-task.
5. While the rightTask is computed directly, the leftTask is allowed to execute in parallel.
6. Once both sub-tasks are complete, the leftTask and rightTask results are joined together, effectively combining the sub-task results to obtain the final result.

**Note**: This divide-and-conquer approach continues recursively until all sub-tasks are small enough to be computed directly, and their results are combined to produce the overall result.

* Concurrency in Java is made simpler and more efficient with the ExecutorService and Fork/Join Framework. The **ExecutorService manages threads**, making it easier to execute tasks concurrently, while the **Fork/Join Framework is designed for parallel processing/programming**, allowing you to divide and conquer complex tasks.
* These tools are valuable additions to a Java developer’s toolbox, enabling the efficient use of multiple cores and improving application performance. By mastering these frameworks, you can build high-performance, multi-threaded Java applications with ease.
* Whether you’re working on a multi-threaded server application, a data processing pipeline, or any other project that requires concurrency, understanding these concepts will undoubtedly help you make the most of Java’s powerful capabilities.

**CompletableFuture:**

Reference link: <https://salithachathuranga94.medium.com/completablefuture-in-java-97b0b392657>



* It’s actually about how to do asynchronous programming with Java.
* You may have already heard of Multi threading with Java.
* That is the base of the asynchronous behaviors in Java.
* But I’m not going to explain threads, but I will explain stuff beyond that.
* Let’s say an advanced version of thread executions with Java.
* **CompletableFuture class was introduced in Java 8.**
* Until then Java had only Future class which came i**n Java 5.**

**Synchronous Vs Asynchronous Programming**

* In Synchronous programming one task is executed at a time.
* After completion of that task, next task will be executed.
* So, it’s having blocking code.
* But in Asynchronous programming multiple tasks are executed at the same time simultaneously based **on thread availability.**
* Simply it’s non blocking code since we are not waiting one task to finish to start the next.

**Futures Vs CompletableFutures:**

* I mentioned that we had Future before CompletableFuture came into picture.
* Future class represents a future result of asynchronous computation.
* It will have a result in future after completion.

Let’s take a simple code example.

|  |
| --- |
| public class FutureExample {  public static void main(String[] args) throws ExecutionException, InterruptedException {  ExecutorService executorService = Executors.newFixedThreadPool(5);  Future<String> f1 = executorService.submit(getCallable("Task 1"));  Future<String> f2 = executorService.submit(getCallable("Task 2"));  Future<String> f3 = executorService.submit(getCallable("Task 3"));  String s1 = f1.get();  System.out.println(s1);  String s2 = f2.get();  System.out.println(s2);  String s3 = f3.get();  System.out.println(s3);  executorService.shutdown();  }  private static Callable<String> getCallable(String taskName) {  return () -> "Task:::" + taskName + " => Thread:::" + Thread.currentThread().getName();  }  }  // output:  // Task:::Task 1 => Thread:::pool-1-thread-1  // Task:::Task 2 => Thread:::pool-1-thread-2  // Task:::Task 3 => Thread:::pool-1-thread-3 |

**Explanation:**

* Here, ExecutorService is a way of binding a thread pool into our java program. There are bunch of pools defined in Java.
* I took newFixedThreadPool for simplicity. I have setup the pool with **5** threads.
* We can pass either Runnable or Callable to submit method.
* Since **future.get()** can be used to retrieve data, I used a callable(Callable will return something but Runnable does not).
* Simply **f1, f2, f3** will be executed together. But when we call future.get()— it’s a blocking call and it will wait until the result completion of our future.
* This way we had some sort/draw of async programming till Java 8.

**Drawbacks had in Future**

* **We can not complete a future manually —** Let’s say we call an API. Due to an issue we get an error. We need to return a cached response in that case. We can not do this with future.
* **Multiple futures can not be chained together —** futures can not be chained or combined in a way where one future result is dependent on previous future result.
* **No exception handling** — there is no proper way to deal with exception situations with futures.
* **Blocking** — future.get() method will block the thread. So, it’s not completely asynchronous.

**CompletableFuture in Java 8**

Let’s start discussing the core stuff. CompletableFuture class is implementing both Future and CompletionStage interfaces.

|  |
| --- |
| public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {  ...... ...... ......  }  Note: It’s an extension of Future. It has a lot of methods where we can create, run, combine, chain multiple futures together and has a very descriptive way of error handling also. |

Explanation:

* When we use this class, behind the scene it’s like we are delegating the tasks into several threads. It actually uses the **global ForkJoinPool => commonPool** to execute the tasks in parallel. If we want, we can pass our own thread pool also.

Creating a CompletableFuture:

* You can create a CompletableFuture simply by using the following no-arg constructor -

CompletableFuture<String> completableFuture = new CompletableFuture<String>();

* This is the simplest CompletableFuture that you can have. All the clients who want to get the result of this CompletableFuture can call CompletableFuture.get() method -

String result = completableFuture.get()

* The get() method blocks until the Future is complete. So, the above call will block forever because the Future is never completed.
* You can use CompletableFuture.complete() method to manually complete a Future -completableFuture.complete("Future's Result")
* All the clients waiting for this Future will get the specified result. And, Subsequent calls to completableFuture.complete() will be ignored.

Let’s discover CompletableFuture class methods:

1. **runAsync:**

* This method takes a Runnable as an argument and returns nothing. It’s a Void asynchronous method. The processing is done by a separate thread in the ForkJoinPool.commonPool().
* Overload methods using runAsync():

|  |
| --- |
| public static CompletableFuture<Void> runAsync(Runnable runnable) {  return asyncRunStage(ASYNC\_POOL, runnable);  }  public static CompletableFuture<Void> runAsync(Runnable runnable,  **Executor executor) {**  return asyncRunStage(screenExecutor(executor), runnable);  } |

|  |
| --- |
| public class RunAsyncExample {  public static void main(String[] args) {  Runnable runnable1 = () -> {  System.out.println("Hello from Task 1::" + Thread.currentThread().getName());  };  CompletableFuture<Void> taskCompletableFuture1 = CompletableFuture.runAsync(runnable);  System.out.println("Hello from Main::" + Thread.currentThread().getName());  taskCompletableFuture1.join();  Runnable runnable2 = () -> {  System.out.println("Hello from Task 2::" + Thread.currentThread().getName());  };  ExecutorService executorService = Executors.newCachedThreadPool();  CompletableFuture<Void> taskCompletableFuture2 = CompletableFuture.runAsync(runnable2, executorService);  taskCompletableFuture2.join();  executorService.shutdown();  }  }  // output:  // Hello from Main::main  // Hello from Task 1::ForkJoinPool.commonPool-worker-1  // Hello from Task 2::pool-1-thread-1 |

**Explanation:**

You can see the printed thread name right? We can see 3 kind of threads! What are they?

* main — Main thread which runs our application
* ForkJoinPool.commonPool-worker-1 — First task has been executed with the **ForkJoinPool thread which is the default thread for CompletableFuture**
* pool-1-thread-1 — Second task execution has an additional parameter. That’s a thread pool I have defined. So, second task is not using the default thread pool anymore. It uses the cached thread pool we created.

**Join method**:

* It is an instance method of the CompletableFuture class.
* It is used to return the value when the future is complete or throws an unchecked exception if completed exceptionally.
* If the task involved in the completion of the CompletableFuture raises an exception, then this method throws a **CompletionException** with the underlying exception as its cause.

1. **supplyAsync:**

* This method takes a Supplier as an argument and returns CompletableFuture of expected result data type. The processing is done by a separate thread in the ForkJoinPool.commonPool().
* If you need to learn more about Java 8 suppliers, please refer here: <https://salithachathuranga94.medium.com/consumer-and-supplier-in-java-8-ec8cf2aea9cf>
* Overlaod methods for supplyAsyn() method.

|  |
| --- |
| public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier) {  return asyncSupplyStage(ASYNC\_POOL, supplier);  }  public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,  **Executor executor) {**  return asyncSupplyStage(screenExecutor(executor), supplier);  } |

Let’s try out a sample code.

|  |
| --- |
| public class SupplyAsyncExample {  public static void main(String[] args) {  Supplier<String> supplier = () -> {  System.out.println("Hello from Task 1::" + Thread.currentThread().getName());  return "Hello from Task 1::" + Thread.currentThread().getName();  };  CompletableFuture<String> taskCompletableFuture = CompletableFuture.supplyAsync(supplier);  System.out.println("Hello from Main::" + Thread.currentThread().getName());  String value = taskCompletableFuture.join();  System.out.println("Value 1::" + value);  Supplier<String> supplier2 **= () -> {**  **System.out.println("Hello from Task 2::" + Thread.currentThread().getName());**  **return "Hello from Task 1::" + Thread.currentThread().getName();**  **};**  ExecutorService executorService = Executors.newCachedThreadPool();  CompletableFuture<String> taskCompletableFuture2 = CompletableFuture.supplyAsync(supplier2, executorService);  String value2 = taskCompletableFuture2.join();  System.out.println("Value 2::" + value2);  executorService.shutdown();  }  }  // output:  // Hello from Main::main  // Hello from Task 1::ForkJoinPool.commonPool-worker-1  // Value 1::Hello from Task 1::ForkJoinPool.commonPool-worker-1  // Hello from Task 2::pool-1-thread-1  // Value 2::Hello from Task 1::pool-1-thread-1 |

* Always the CompletableFuture is running on a new worker thread.
* Main thread will execute independently without bothering whether the other asynchronous operations are completed or not. I have used our own thread pool for second task.
* There are more things can be done with these futures. They will be explained below.

**Java – CompletableFuture class: join() vs get():**

* join() is defined in CompletableFuture whereas get() comes from interface Future
* join() throws unchecked exception whereas get() throws checked exceptions
* You can interrupt get() and then throws an InterruptedException
* get() method allows to specify the maximum wait time

|  |
| --- |
| **Important Note:** when you call get() or join() method on current execution thread, that will block/wait until itget output from future or completable future.  **Example Program:**  CompletableFuture<String> taskCompletableFuture = CompletableFuture.supplyAsync(() -> {  System.out.println("Hello from Task 1::supplyAsync::" + Thread.currentThread().getName());  return "Hey";  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  CompletableFuture<String> stringCompletableFuture = taskCompletableFuture.thenApplyAsync(data -> {  try {  System.exit(0);  Thread.sleep(50000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  System.out.println("Hello from Task 1::thenApplyAsync::" + Thread.currentThread().getName());  return data + " Developers!";  });    System.out.println("Hello before join method");  String result = stringCompletableFuture.join();//main thread will block  System.out.println("Hello after join method");//It print this statement after getting result from thenApplyAsync()  System.out.println(result); |

**Is CompletableFuture Non-blocking?**

* Why did we need CompletableFurure in the first place? Using the Future interface, we could only retrieve the result by calling the get() method. However, this method represents a blocking operation. In other words, it’ll block the current thread until the result of the task becomes available.
* If we need to perform additional actions on the result, we’ll end up with blocking operations.On the other hand, thanks to CompletionStage, CompletableFuture provides the ability to chain multiple computations together that can run concurrently. This functionality allows us to create a chain of tasks where the next task is triggered when the current task is completed.
* Furthermore, we can specify what should happen once we get the result from the future without blocking the current thread.
* The CompletableFuture class represents both the stage in dependent processes, where one stage’s completion triggers another, and its result.

**Blocking vs. Non-blocking**

Next, let’s understand the difference between blocking and non-blocking processing.

In the blocking operation, the calling thread waits until the operation in another thread completes before continuing with its execution:

Here, the tasks execute sequentially. Thread 1 is blocked by Thread 2. In other words, Thread 1 can’t continue with its execution until Thread 2 finishes processing its tasks.

We can look at the blocking processing as synchronous operations.

However, blocking operations in our system can cause performance issues, especially in applications that require high availability and scalability.

In contrast, a non-blocking operation allows threads to perform multiple computations simultaneously without having to wait for each task to complete.

The current thread can continue with its execution while the other threads perform tasks in parallel:

**CompletableFuture and Non-blocking Operations**

The main advantage of using CompletableFuture is its ability to chain multiple tasks together that will be executed without blocking the current thread. Therefore, we can say the CompletableFuture is non-blocking.

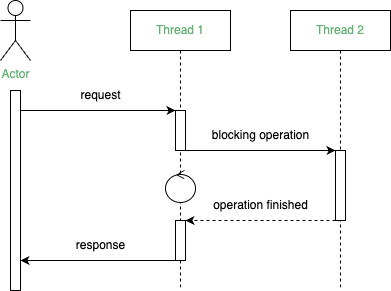
Additionally, it provides several methods that allow us to perform tasks in a non-blocking way, including:

* supplyAsync(): executes a task asynchronously and returns a CompletableFuture representing the result
* thenApply(): applies a function to the result of a previous task and returns a CompletableFuture representing the transformed result
* thenCompose(): executes a task that returns a CompletableFuture and returns a CompletableFuture representing the result of the nested task
* allOf(): executes several tasks in parallel and returns a CompletableFuture representing the completion of all tasks

**When Is CompletableFuture Blocking?**

* Although CompletableFuture is used to perform non-blocking operations, it can still end up blocking the current thread in certain scenarios.
* In asynchronous communication, we usually have a callback mechanism to retrieve the result of the computation. However, CompletableFuture doesn’t notify us upon its completion.
* If needed, we can retrieve the result in the calling thread using the get() method.
* Nevertheless, we need to be aware the get() method returns the result using blocking processing. If required, it waits for the computation to complete and then returns the result.
* Therefore, we’ll end up blocking the current thread until the future completes:
* Reference link: <https://www.baeldung.com/java-completablefuture-non-blocking#:~:text=Therefore%2C%20we%20can%20say%20the,a%20CompletableFuture%20representing%20the%20result>

|  |
| --- |
| CompletableFuture<String> completableFuture = CompletableFuture .supplyAsync(() -> "Baeldung") .thenApply(String::toUpperCase);  assertEquals("BAELDUNG", completableFuture.get());  CompletableFuture<String> completableFuture = CompletableFuture .supplyAsync(() -> "Blocking") .thenApply(s -> s + " Operation") .thenApply(String::toLowerCase); |

assertEquals("blocking operation", completableFuture.join());

**Java – CompletableFuture | thenApply vs thenCompose completable-futurejava**

Reference link: <https://itecnote.com/tecnote/java-completablefuture-thenapply-vs-thencompose/>

From the Java docs:

* thenApply(Function<? super T,? extends U> fn)
* Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function.
* thenCompose(Function<? super T,? extends CompletionStage<U>> fn)
* Returns a new CompletionStage that, when this stage completes normally, is executed with this stage as the argument to the supplied function.
* I get that the 2nd argument of thenCompose extends the CompletionStage where thenApply does not.
* Could someone provide an example in which case I have to use thenApply and when thenCompose?

thenApply is used if you have a **synchronous mapping function.(Like map() method in Java Streams or Optional)**

**Note: Until it et result of completable future it will not execute thenApply method functionality.**

|  |
| --- |
| CompletableFuture<Integer> future =  CompletableFuture.supplyAsync(() -> 1)  .thenApply(x -> x+1); |

thenCompose is used if you have **an asynchronous mapping function** (i.e. one that returns a CompletableFuture). It will then return a future with the result directly, rather than a nested future.

It is like flatMap() in java streams or Optional in java

|  |
| --- |
| CompletableFuture<Integer> future =  CompletableFuture.supplyAsync(() -> 1)  .thenCompose(x -> CompletableFuture.supplyAsync(() -> x+1)); |

What is CompletableFuture.thenCombine() in Java?(like reduce() method in java streams or Optional )

**Parameters**

CompletionStage<? extends U> other: This is the other completable future that needs to be completed.

BiFunction<? super T,? super U,? extends V> fn: This is the BiFunction that needs to be executed.

**Return value**

This method returns a new CompletableFuture.

Example:

|  |
| --- |
| import java.util.concurrent.\*;  import java.util.function.BiFunction;  public class Main {  static void sleep(int millis){  try {  Thread.sleep(millis);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  static void executionThread(){  System.out.println("Thread execution - " + Thread.currentThread().getName());  }  public static void main(String[] args){  CompletableFuture<String> completableFuture1 = CompletableFuture.supplyAsync(() -> {  sleep(1000);  String stringToPrint = "Educative";  System.out.println("----\nsupplyAsync first future - " + stringToPrint);  executionThread();  return stringToPrint;  });  CompletableFuture<String> completableFuture2 = CompletableFuture.supplyAsync(() -> {  sleep(2000);  String stringToPrint = "Edpresso";  System.out.println("----\nsupplyAsync second future - " + stringToPrint);  executionThread();  return stringToPrint;  });  BiFunction<String, String, String> stringStringBiFunction = (res1, res2) -> String.format("\"completableFuture1 result - %s | completableFuture2 result - %s\"", res1, res2);  CompletableFuture<String> finalCompletableFuture = completableFuture1.thenCombine(completableFuture2, stringStringBiFunction);  System.out.println("Output of thenCombine - " + finalCompletableFuture.join());  sleep(3000);  }  } |

**Note**: We can pass already computed completable future reference as return type of this thencompose method , that completable future is started asynchronously.

**Callback Methods:**

* These methods are mainly used to chain set of futures and do whatever we want according to the scenario addressed.

|  |
| --- |
| Examples:  ThenApplyAsync, ThenAcceptAsync, ThenAcceptAsync, ThenRunAsync, ThenComposeAsync, ThenCombineAsync |

**NOTE**:

Each of this kind of method has 3 versions. For an example,

1. **thenApply(fn)** — runs fn on a thread defined by the CompleteableFuture on which it is called, so you generally cannot know where this will be executed. It might immediately execute if the result is already available.
2. **thenApplyAsync(fn)** — runs fn on a environment-defined executor regardless of circumstances. For CompletableFuture this will generally be ForkJoinPool.commonPool().
3. **thenApplyAsync(fn,exec)** — runs fn on given executor instead of ForkJoinPool executor.

* The main difference in these 3 versions will be how they gain thread control and execute — on which thread it will be executed.
* But remember! There is nothing in **thenApplyAsync** that is more asynchronous than **thenApply** from the contract of these methods.
* Both does the same job.

I will show it on next section 😎.

1. **thenApplyAsync:**

This method takes a Function as an argument and returns CompletableFuture of transformed data. As an example, if you want to modify a string returned from a previous future and return, you can use this method. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Let’s understand with an example.

**Example**:

|  |
| --- |
| public class ThenApplyExample {  public static void main(String[] args) {  CompletableFuture<String> taskCompletableFuture = CompletableFuture.supplyAsync(() -> {  System.out.println("Hello from Task 1::supplyAsync::" + Thread.currentThread().getName());  return "Hey";  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  CompletableFuture<String> stringCompletableFuture = taskCompletableFuture.thenApplyAsync(data -> {  System.out.println("Hello from Task 1::thenApplyAsync::" + Thread.currentThread().getName());  return data + " Developers!";  });  String result = stringCompletableFuture.join();  System.out.println(result);  }  }  // output:  // Hello from Main::main  // Hello from Task 1::supplyAsync::ForkJoinPool.commonPool-worker-1  **// Hello from Task 1::thenApplyAsync::ForkJoinPool.commonPool-worker-1**  // Hey Developers! |

**Explanation:**

* I have concatenated two strings using two futures. **The first string was transformed into another**. You can see right?
* There’s one important point there. According to the system logs, it shows that both futures have been executed using the same thread.

**What happens if we use ThenApply instead of ThenApplyAsync?**

* Just change the method name in the above code and see the logs.

|  |
| --- |
| // Hello from Main::main  // Hello from Task 1::supplyAsync::ForkJoinPool.commonPool-worker-1  **// Hello from Task 1::thenApplyAsync::main**  // Hey Developers! |

Explanation:

* It seems same right? Noooo!!! Look at the 3rd log line! Thread name has been changed to main instead of ForkJoinPool thread. 😮
* So, async method was able to gain control, reuse and execute the future on the same thread where previous call was executed. But the other method could not access the same thread. But again, this will depends on how much time taken to execute futures also and how JVM is scheduling the threads on availability. Result of non async method can be varied based on that.
* **All callback methods will have the same behavior for async and non async methods. So, I will show it only here.** Otherwise article will be lengthy. 😉

1. **thenAcceptAsync**

* This method takes a Consumer as an argument and returns nothing. It’s also a Void asynchronous method. The processing is done by a **separate thread in the ForkJoinPool.commonPool().** Let’s understand with an example.

|  |
| --- |
| public class ThenAcceptExample {  public static void main(String[] args) {  CompletableFuture<String> taskCompletableFuture = CompletableFuture.supplyAsync(() -> {  System.out.println("Hello from Task 1::supplyAsync::" + Thread.currentThread().getName());  return "Hey";  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  Consumer<String> consumer = (data) -> System.out.println(data + " Developers! Hello from Task 1::thenAcceptAsync::" + Thread.currentThread().getName());  taskCompletableFuture.thenAcceptAsync(consumer).join();  }  }  // output:  // Hello from Main::main  // Hello from Task 1::supplyAsync::ForkJoinPool.commonPool-worker-1  // Hey Developers! Hello from Task 1::thenAcceptAsync::ForkJoinPool.commonPool-worker-1 |

* I have printed the data came from previous future with some more data, in the example. Since we are using Async method, we have the executor thread to execute the futures.
* PS: Usually a Consumer takes something and do some operation without returning anything. If you need to learn more about Java 8 consumers, please refer here: <https://salithachathuranga94.medium.com/consumer-and-supplier-in-java-8-ec8cf2aea9cf>

1. **thenRunAsync**

* This method takes a Runnable as an argument and returns nothing. It’s also a Void asynchronous method. The processing is done by a separate thread in the ForkJoinPool.commonPool(). Let’s understand with an example.

|  |
| --- |
| public class ThenRunExample {  public static void main(String[] args) {  CompletableFuture<String> taskCompletableFuture = CompletableFuture.supplyAsync(() -> {  System.out.println("Hello from Task 1::supplyAsync::" + Thread.currentThread().getName());  return "Hey";  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  Runnable runnable = () -> System.out.println("Finishing Task 1::thenRunAsync::" + Thread.currentThread().getName());  **taskCompletableFuture.thenRunAsync(runnable).join();**  }  }  // output:  // Hello from Task 1::supplyAsync::ForkJoinPool.commonPool-worker-1  // Hello from Main::main  // Finishing Task 1::thenRunAsync::ForkJoinPool.commonPool-worker-1  We can use this method to run any void method or print statement after futures has been completed. |

1. **thenComposeAsync**

* This method takes a Function as an argument and returns CompletableFuture of the expected result. It is used to chain two dependent futures sequentially. Let’s understand with an example.
* **Scenario**: We get customer information from an API and using that info we are calling payments API to get relevant payment details. Two calls are inter dependent.

Example:

|  |
| --- |
| public class ThenComposeExample {  private static void sleep(int seconds) {  try {  TimeUnit.SECONDS.sleep(seconds);  } catch (InterruptedException e) {  throw new RuntimeException(e);  }  }  private static CompletableFuture<Map<String, String>> getUserDetails() {  return CompletableFuture.supplyAsync(() -> {  sleep(5);  System.out.println("getUserDetails:::" + Thread.currentThread().getName());  return getUser();  });  }  private static Map<String, String> getUser() {  Map<String, String> user = new ConcurrentHashMap<>();  user.put("userId", "1234");  user.put("userName", "salitha");  user.put("phoneNo", "0777123456");  return user;  }  private static CompletableFuture<List<String>> getPayments(String userName) {  return CompletableFuture.supplyAsync(() -> {  sleep(7);  System.out.println("getPayments:::" + Thread.currentThread().getName());  return Arrays.asList(  "USER: " + userName + " => $100",  "USER: " + userName + " => $65"  );  });  }  public static void main(String[] args) {  long startTime = System.currentTimeMillis();  CompletableFuture<List<String>> paymentsFuture = getUserDetails()  .thenComposeAsync(userData -> {  return getPayments(userData.get("userName"));  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  sleep(4);  List<String> payments = paymentsFuture.join();  System.out.println(payments);  long endTime = System.currentTimeMillis();  System.out.println("Time taken::" + (endTime - startTime) / 1000);  }  }  // output:  // getUserDetails:::ForkJoinPool.commonPool-worker-1  // Hello from Main::main  // getPayments:::ForkJoinPool.commonPool-worker-2  // [USER: salitha => $100, USER: salitha => $65]  // Time taken::12  **Explanation**:  If you see the logs, we have run the two tasks getUserDetails and getPayments methods on separate threads. Here, worker-1 and worker-2 are the ForkJoinPool threads allocated. Even though we have use sleep for 5 + 7 + 4 = 16 seconds, it has been executed within 12 seconds resulting asynchronous behavior. |

1. **thenCombineAsync**:

* This method takes a BiFunction as an argument and returns a CompletableFuture of the expected result. It is used to combine two independent futures parallel and combine some result. Let’s understand with an example.
* Scenario: We need to get a user email and weather report parallel and send an email to that user.

**Example**:

|  |
| --- |
| public class ThenCombineExample {  private static void sleep(int seconds) {  try {  TimeUnit.SECONDS.sleep(seconds);  } catch (InterruptedException e) {  throw new RuntimeException(e);  }  }  private static CompletableFuture<String> getWeather() {  return CompletableFuture.supplyAsync(() -> {  sleep(5);  System.out.println("getUserDetails:::" + Thread.currentThread().getName());  return "Sunny, Temperature: 28C";  });  }  private static CompletableFuture<String> getUserEmail() {  return CompletableFuture.supplyAsync(() -> {  sleep(5);  System.out.println("getUserEmail:::" + Thread.currentThread().getName());  return "john@gmail.com";  });  }  public static void main(String[] args) {  long startTime = System.currentTimeMillis();  CompletableFuture<String> weatherEmailFuture = getUserEmail()  .thenCombineAsync(getWeather(), (email, weather) -> {  System.out.println("Sending email to:::" + email + " Weather report => " + weather);  System.out.println("Sending email:::" + Thread.currentThread().getName());  return email + " => " + weather;  });  System.out.println("Hello from Main::" + Thread.currentThread().getName());  sleep(4);  String email = weatherEmailFuture.join();  System.out.println(email);  long endTime = System.currentTimeMillis();  System.out.println("Time taken::" + (endTime - startTime) / 1000);  }  }  // Hello from Main::main  // getUserEmail:::ForkJoinPool.commonPool-worker-1  // getUserDetails:::ForkJoinPool.commonPool-worker-2  // Sending email to:::john@gmail.com Weather report => Sunny, Temperature: 28C  // Sending email:::ForkJoinPool.commonPool-worker-1  // john@gmail.com => Sunny, Temperature: 28C  // Time taken::5  **Explanation:**  Following the same pattern, getUserEmail and getWeather methods have been executed on two separate threads. And it has reused the thread — worker-1 which was released. Even though we have used sleep for 5 + 5 + 4 = 14 seconds, it has been executed within 5 seconds with a great asynchronous behavior. |

**Aggregation Methods:**

* Please mind this word — **aggregation is used by me just to differentiate methods. It’s not a standard classification! 😅**
* So far you know, we have several methods to deal with **two futures, right**? But how to deal with **more than two futures? What are the methods available for this**?

|  |
| --- |
| allOf and anyOf methods |

1. **allOf()**

* When you have multiple futures to deal and perform some action after all those futures are completed only, this is the method to use.
* It returns a new CompletableFuture object when all of the specified CompletableFutures are complete. So, this method accepts a list of completablefutures.
* We can define and run them parallel and provide the futures into allOf method. If any of the specified CompletableFutures is completed with an exception, the resulting CompletableFuture does as well, with a CompletionException as the cause. Let’s understand with an example

**Example**:

|  |
| --- |
| public class AllOfExample {  private static void sleep(int seconds) {  try {  TimeUnit.SECONDS.sleep(seconds);  } catch (InterruptedException e) {  throw new RuntimeException(e);  }  }  private static CompletableFuture<String> futureOne() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureOne::" + Thread.currentThread().getName());  sleep(4);  return "CF1";  });  }  private static CompletableFuture<String> futureTwo() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureTwo::" + Thread.currentThread().getName());  sleep(3);  return "CF2";  });  }  private static CompletableFuture<String> futureThree() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureThree::" + Thread.currentThread().getName());  sleep(2);  return "CF3";  });  }  public static void main(String[] args) {  long startTime = System.currentTimeMillis();  List<CompletableFuture<String>> completableFutures = Arrays.asList(futureOne(), futureTwo(), futureThree());  CompletableFuture<Void> future = CompletableFuture.allOf(completableFutures.toArray(new CompletableFuture[0]));  System.out.println("Hello from Main::" + Thread.currentThread().getName());  sleep(6);  CompletableFuture<List<String>> allFutureResults = future  .thenApply(t -> completableFutures.stream().map(CompletableFuture::join)  .collect(Collectors.toList()));  System.out.println("Result: " + allFutureResults.join());  long endTime = System.currentTimeMillis();  System.out.println("Time taken::" + (endTime - startTime) / 1000);  }  }  // output:  // Hello from Main::main  // futureTwo::ForkJoinPool.commonPool-worker-2  // futureOne::ForkJoinPool.commonPool-worker-1  // futureThree::ForkJoinPool.commonPool-worker-3  // Result: [CF1, CF2, CF3]  // Time taken::6  **Explanation**:  The futures I have defined has been executed on 3 separate worker threads as per the logs. And if you execute the code, you will see they are executed parallel and not one after the other. Even though we have used sleep for 4 + 3 + 2 + 6 = 15 seconds, it has been executed within 6 seconds. That’s how asynchronous code behaves. |

1. **anyOf()**

* When you have several asynchronous tasks and you want to return a result as soon as 1 future is completed, this is the method you would choose. The anyOf() method gets tricky when a list of completableFutures return multiple types of outcomes. Due to this, the user is not able to tell which future got completed first. Let’s understand with an example.

**Example:**

|  |
| --- |
| public class AnyOfExample {  private static void sleep(int seconds) {  try {  TimeUnit.SECONDS.sleep(seconds);  } catch (InterruptedException e) {  throw new RuntimeException(e);  }  }  private static CompletableFuture<String> futureOne() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureOne::" + Thread.currentThread().getName());  sleep(4);  return "CF1";  });  }  private static CompletableFuture<String> futureTwo() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureTwo::" + Thread.currentThread().getName());  sleep(3);  return "CF2";  });  }  private static CompletableFuture<String> futureThree() {  return CompletableFuture.supplyAsync(() -> {  System.out.println("futureThree::" + Thread.currentThread().getName());  sleep(2);  return "CF3";  });  }  public static void main(String[] args) {  long startTime = System.currentTimeMillis();  List<CompletableFuture<String>> completableFutures = Arrays.asList(futureOne(), futureTwo(), futureThree());  CompletableFuture<Object> future = CompletableFuture.anyOf(completableFutures.toArray(new CompletableFuture[0]));  System.out.println("Hello from Main::" + Thread.currentThread().getName());  System.out.println(future.join());;  long endTime = System.currentTimeMillis();  System.out.println("Time taken::" + (endTime - startTime) / 1000);  }  }  // output:  // futureTwo::ForkJoinPool.commonPool-worker-2  // futureOne::ForkJoinPool.commonPool-worker-1  // Hello from Main::main  // futureThree::ForkJoinPool.commonPool-worker-3  // CF3  // Time taken::2  **Explanation**:  Let’s understand the logs first. As always, futures are executed on separate threads. But what is the result giving by future.join()? It is “CF3”. How it has happened? If you see the 3 completablefuture methods, futureThree method has the less time of execution since I have slept it for 2 seconds. As soon as it is completed, we have the result! That’s why we got “CF3”. |

**Handling Exceptions:**

There are 3 ways to handle exceptions while executing completable futures. Let’s look into them.

**1️. handle:**

**Takes a BiFunction** — result and exception which is executed when the stage completes either successfully or exceptionally. It does not matter whether program is executed properly or not. I will quickly show how it works using an erroneous scenario.

|  |
| --- |
| public class HandleExample {  public static void main(String[] args) {  CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {  int x = 10;  return x / 0;  }).handle((result, error) -> {  if (error != null) {  System.out.println("Error occurred!: " + error.getMessage());  return 0;  }  return result;  });  System.out.println(future.join());;  }  }  // output:  // Error occurred!: java.lang.ArithmeticException: / by zero  // 0 |

Explanation:

This future will definitely throw an Exception: Arithmetic Exception since I’m trying to divide by zero. So, handle method will catch that exception in the BiFunction and we can log the error message. If error is NOT NULL, we are returning zero, Otherwise result from our future. Since we have result and error at the same time, we have to perform a NULL check for one item. If I change the statement by replacing with this => return x / 2, then you will see the result as 5.

We can modify this example by chaining a callback method.

|  |
| --- |
| public class HandleExample {  public static void main(String[] args) {  CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {  int x = 10;  return x / 2;  }).handle((result, error) -> {  if (error != null) {  System.out.println("Error occurred!: " + error.getMessage());  return 5;  }  return result;  }).thenApplyAsync(x -> x + 20);  System.out.println(future.join());  }  }  // output:  // Error occurred!: java.lang.ArithmeticException: / by zero  // 25 |

* We can modify this example by chaining a callback method.
* Output is 25 since return 5 when there’s an exception. So, thenApplyAsync will take the result and add 20 and return.
* This is one way that we can handle exceptions occurs while futures are executed.

**2. exceptionally:**

Takes a Function — exception which is executed when the stage completes exceptionally. We will only get the error and not the result as before. I will use the same scenario above to demonstrate this method.

**Example**:

|  |
| --- |
| public class ExceptionallyExample {  public static void main(String[] args) {  CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {  int x = 10;  return x / 0;  }).exceptionally(error -> {  System.out.println("Error occurred!: " + error.getMessage());  return 0;  });  System.out.println(future.join());  }  }  // output:  // Error occurred!: java.lang.ArithmeticException: / by zero  // 0 |

Here, we only have the access to the exception. It does not catch the result. Output will be same as above. Both methods are catching exceptions, but in different ways.

We can modify this example also by chaining a callback method.

|  |
| --- |
| public class ExceptionallyExample {  public static void main(String[] args) {  CompletableFuture.supplyAsync(() -> {  int x = 10;  return x / 0;  }).exceptionally(error -> {  System.out.println("Error occurred!: " + error.getMessage());  return 0;  }).thenAcceptAsync(x -> {  System.out.println(x + 10);  });  }  }  // output:  // Error occurred!: java.lang.ArithmeticException: / by zero  // 10  **Explanation**:  Output became 10 since return zero when there’s an exception. So, thenAcceptAsync will take the result and add 10 to it and print. |

**3.whenComplete**

This also takes a BiFunction — result and exception which is executed when the stage completes either successfully or exceptionally.

|  |
| --- |
| public class WhenCompleteExample {  public static void main(String[] args) {  CompletableFuture.supplyAsync(() -> {  int x = 10;  return x / 2;  }).whenComplete((result, error) -> {  if (error != null) {  System.out.println("Error occurred!: " + error.getMessage());  } else {  System.out.println(result);  }  });  }  }  // output:  // Error occurred!: java.lang.ArithmeticException: / by zero |

**If we chain a callback method after this, it won’t behave like exceptionally and handle methods. We cannot return a result a value inside this whenComplete method. That’s the reason.**

**Important Note:**

* Asynchronous programming is very important and useful while we write real world applications. For an example, any database call will take some time to execute.
* But if we need to execute an API call without depending on the previous DB call? Then we have to delegate tasks into threads. Without going for naïve approach using thread pools and executors, we can use CompletableFutures! Otherwise the same thread will be blocked. At the end, user will experience a latency.

**ThreadLocal**:

* In the world of multi-threaded Java applications, managing data that is unique to each thread can be a challenging task.
* Java’s ThreadLocal class comes to the rescue as a powerful tool for handling thread-specific data.
* It provides a straightforward mechanism to store and access data in a way that ensures thread safety.
* In this comprehensive article, we will delve into the ThreadLocal class in Java, exploring its purpose, when to use it, when not to use it, and providing specific examples for each scenario.

**What is ThreadLocal?**

* ThreadLocal is a class in the java.lang package that allows you to create thread-local variables.
* These variables are unique to each thread, which means that each thread accessing a ThreadLocal variable gets its own independent copy.
* This isolation eliminates synchronization overhead and contention issues, making it particularly useful in multi-threaded environments.

**How ThreadLocal Works**

* Under the hood, ThreadLocal uses a map-like data structure that associates a value with the current thread.
* When you set a value using ThreadLocal.set(), it is stored in this thread-specific map.
* Retrieving the value with ThreadLocal.get() always returns the value associated with the current thread.
* **One map per one ThreadLocal object that can be accessed through current thread only.**

**When to Use ThreadLocal**

* A frequent scenario arises when you are dealing with an object that is not thread-safe, but you aim to avoid the need for synchronization when accessing that object.
* In this case, the solution is to provide each thread with its own instance of the object.

**Use Case 1: Storing User Session Data**

* ThreadLocal is often used in web applications to store user session data.
* Each user’s session information can be stored in a ThreadLocal variable, ensuring that it is easily accessible throughout the request-handling process without worrying about thread interference.

**Here’s a simplified example:**

|  |
| --- |
| package com.threadlocal;  public class MyClass {  private static final ThreadLocal<User> threadLocal = new ThreadLocal<>();  private static final ThreadLocal threadLocal1 = new ThreadLocal<>();  public static void main(String[] args) {  getLoggedInUser();  try {  doSomething();  doSomethingElse();  renderResponse();  } finally {  threadLocal.remove();  }  }  private static void getLoggedInUser() {  // Retrieve the logged-in user  User user = retrieveUser();  threadLocal.set(user);  }  private static User retrieveUser() {  // TODO Auto-generated method stub  return new User();  }  private static void doSomething() {  User user = threadLocal.get();  // Access the user object  }  private static void doSomethingElse() {  User user = threadLocal.get();  // Access the user object  }  private static void renderResponse() {  User user = threadLocal.get();  // Access the user object  }  }  class User {    } |

* In this example, the User object is stored in the ThreadLocal variable threadLocal . The getLoggedInUser() method retrieves the logged-in user.
* Then, the threadLocal.set(user) statement sets the user object in the ThreadLocal variable, making it accessible throughout the execution of the
* main() method and any subsequent method calls.In each method that needs access to the user object, the threadLocal.get()
* method is called to retrieve the user object specific to the current thread.This allows each method to access the user object without the need for
* passing it as a parameter.
* It’s important to note that the remove() method should be called in a finally block to ensure the cleanup of the ThreadLocal variable after it is no longer needed. This helps prevent memory leaks, especially in environments that use thread pools.

|  |
| --- |
| public class UserSession {  private static final ThreadLocal<User> userSession = ThreadLocal.withInitial  public static User getCurrentUser() {  return userSession.get();  }  } |

**When Not to Use ThreadLocal**

**Case 1: Excessive Memory Consumption**

* Using ThreadLocal for too many objects can lead to **excessive memory consumption,** especially if the objects are large.
* It’s crucial to consider the **memory impact** when using ThreadLocal.

**Case 2: Shared Resources**

* ThreadLocal is not suitable for sharing resources among threads.
* If you need shared resources, consider other synchronization mechanisms such as locks, semaphores, or concurrent collections.

|  |
| --- |
| public class SharedResource {  private static final ThreadLocal<Resource> resourceThreadLocal = ThreadLocal  // This is not the correct use of ThreadLocal for shared resources  } |

**Use remove()function of ThreadLocal**

* When you use ThreadLocal in application servers with thread pools, you can run into an issue called classloading leaks. This happens because ThreadLocal references data within a specific thread. To avoid problems, it’s
* crucial to properly clean up any ThreadLocals you use by using theThreadLocal’s remove() method.
* If you forget to clean up, any references held by ThreadLocal to classes loaded as part of a web application will stay in memory permanently and won’t be garbage collected. Even if you redeploy or undeploy the web application, the Thread’s reference to your web app’s class(es) won’t be cleaned up because the Thread is not owned by your web app.
* Consequently, each redeployment creates a new class instance that never gets garbage collected.
* This situation can lead to out-of-memory errors, specifically **java.lang.OutOfMemoryError: PermGen space**.
* People often try to fix this by increasing the -XX:MaxPermSize setting instead of addressing the underlying issue.
* In conclusion, the ThreadLocal class in Java is a valuable tool for managing thread-specific data in multi-threaded applications. When used appropriately, it can significantly simplify code and improve performance by avoiding synchronization overhead.
* However, it should be used judiciously, considering factors like memory consumption and the nature of shared resources. By understanding when to use and when not to use ThreadLocal, developers can harness its power effectively to build robust and efficient multi-threaded Java applications.

**CountDownLatch and CyclicBarrier in Java:**

CountDownLatch and CyclicBarrier are two classes in the Java concurrency library that are used to synchronize the execution of multiple threads. They both have different use cases and are used to achieve different goals, but they share the same basic principle of blocking the execution of a thread until a certain condition is met.

**CountDownLatch**

* CountDownLatch is a synchronization aid that allows one or more threads to wait for a set of operations to complete. It is initialized with a count, and each time the count is decremented by one, the waiting threads are released. Once the count reaches zero, all threads are released and can continue their execution.
* Let’s take an example of a thread that needs to wait for a set of other threads to complete before it can continue its execution. We can use a CountDownLatch to achieve this.

|  |
| --- |
| public class CountDownLatchExample {  public static void main(String[] args) throws InterruptedException {  CountDownLatch latch = new CountDownLatch(3);  Thread t1 = new Thread(() -> {  // do some work  latch.countDown();  });  Thread t2 = new Thread(() -> {  // do some work  latch.countDown();  });  Thread t3 = new Thread(() -> {  // do some work  latch.countDown();  });  t1.start();  t2.start();  t3.start();  // wait for t1, t2, and t3 to complete  latch.await();  // continue execution  }  } |

In this example, we have a main thread that creates three other threads (t1, t2, and t3) and starts them. We also have a CountDownLatch initialized with a count of 3. Each time one of the threads completes its work, it calls the countDown() method on the latch, which decrements the count by one. Once the count reaches zero, the main thread is released and can continue its execution.

**CyclicBarrier**

* CyclicBarrier is another synchronization aid that allows multiple threads to wait for each other to reach a common point before continuing their execution. It is initialized with a count, and each time a thread reaches the barrier, the count is decremented by one. Once the count reaches zero, all threads are released and can continue their execution.
* Let’s take an example of a group of threads that need to work together to solve a problem. We can use a CyclicBarrier to achieve this.

|  |
| --- |
| public class CyclicBarrierExample {  public static void main(String[] args) throws InterruptedException {  CyclicBarrier barrier = new CyclicBarrier(3);  Thread t1 = new Thread(() -> {  // do some work  barrier.await();  // continue execution  });  Thread t2 = new Thread(() -> {  // do some work  barrier.await();  // continue execution  });  Thread t3 = new Thread(() -> {  // do some work  barrier.await();  // continue execution  });  t1.start();  t2.start();  t3.start();  }  } |

**Explanations**:

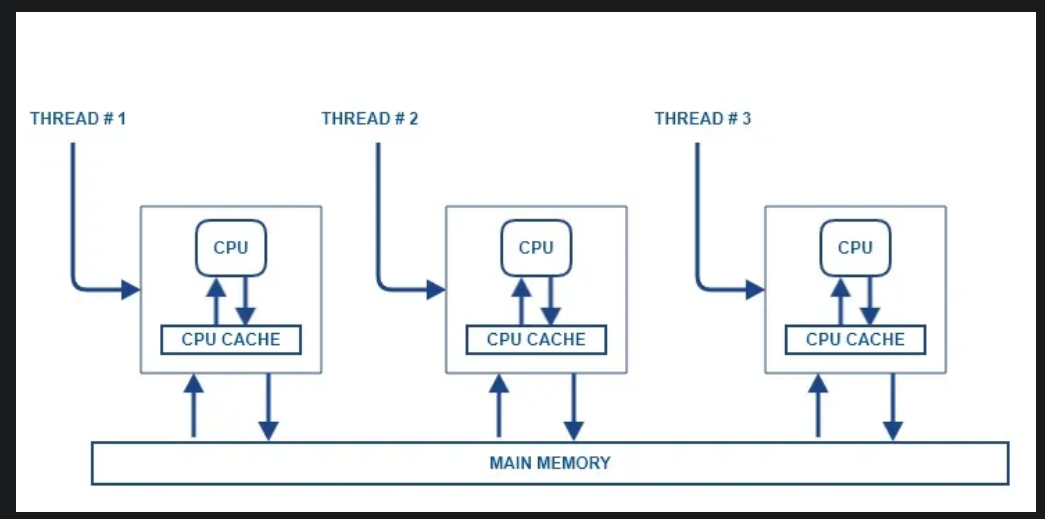
* In this example, we have a main thread that creates three other threads (t1, t2, and t3) and starts them. We also have a CyclicBarrier initialized with a count of 3. Each time one of the threads completes its work, it calls the await() method on the barrier. Once all three threads have reached the barrier, the count reaches zero, and all threads are released and can continue their execution.
* One important thing to note about CyclicBarrier is that it can be reused. Once all threads have been released, the barrier can be reset and used again for another set of threads. This can be useful in situations where the same set of threads need to perform the same task multiple times.

**Difference between CountDownLatch and CyclicBarrier**

* The main difference between CountDownLatch and CyclicBarrier is the way they handle exceptions. CountDownLatch does not provide any mechanism to handle exceptions thrown by the threads that are waiting on the latch. If an exception is thrown by one of the threads, the latch will not be decremented, and the other threads will continue to wait indefinitely.
* CyclicBarrier, on the other hand, provides a way to handle exceptions thrown by the threads. It has a broken state that is set when a thread throws an exception while waiting on the barrier. Once the barrier is broken, all threads that are currently waiting on the barrier will be released and any subsequent threads that try to wait on the barrier will be thrown a **BrokenBarrierException**.
* Additionally, CyclicBarrier allows for specifying a Runnable action to be executed when the barrier is tripped. This action is executed by the last thread to reach the barrier and can be used to perform any necessary cleanup or to initiate the next step in the program.
* Another feature of the CyclicBarrier is that it can be used with a timeout. The await() method of the CyclicBarrier can take a timeout parameter, which specifies how long the thread should wait for other threads to reach the barrier before timing out. If a timeout occurs, the barrier is broken, and the thread that timed out will be thrown a TimeoutException.
* In conclusion, CountDownLatch and CyclicBarrier are two powerful tools in the Java concurrency library that can be used to synchronize the execution of multiple threads. CountDownLatch is a simple, one-time use class that is useful for waiting for a set of operations to complete, while CyclicBarrier is a more advanced class that can be used for coordinating the execution of multiple threads, handling exceptions, and specifying a runnable action to be executed when the barrier is tripped. Both classes can be used to ensure that threads work together in a coordinated and efficient manner, but the choice of which one to use depends on the specific requirements of the program.

In Java, there are three types of variables used for managing concurrent access to shared resources: **atomic, volatile, and synchronized.**

While all three of these variables are used for the same purpose, there are significant differences in how they work and how they should be used.



**Atomic Variables:**

* Atomic variables are used to perform atomic operations on primitive data types such as int, long, double, etc.
* They provide a way to modify the value of the variable atomically (i.e., in one atomic operation), thus avoiding race conditions.
* This means that an atomic variable operation will complete before another operation can start.
* In Java, the java.util.concurrent.atomic package provides atomic variables.
* For example, let’s say we have an integer counter that multiple threads can access concurrently.
* If we use an atomic integer, we can modify the counter atomically, like this:

|  |
| --- |
| import java.util.concurrent.atomic.AtomicInteger;  public class AtomicCounter {  private AtomicInteger counter = new AtomicInteger(0);  public void increment() {  counter.incrementAndGet();  }  public int getCounter() {  return counter.get();  }  } |

* In the above example, the increment() method uses the incrementAndGet() method provided by AtomicInteger to increment the counter atomically.
* Similarly, the getCounter() method returns the value of the counter.

**Volatile Variables:**

* Volatile variables are used to indicate that a variable’s value may be modified by multiple threads. When a variable is declared volatile, the compiler and the JVM ensure that any read or write operation on that variable is done directly from or to main memory, rather than from a thread’s local cache.
* This ensures that any changes to the variable’s value are immediately visible to other threads.
* For example, let’s say we have a boolean flag that multiple threads can access concurrently. If we use a volatile boolean, we can ensure that any changes to the flag’s value are immediately visible to other threads, like this:

|  |
| --- |
| public class VolatileFlag {  private volatile boolean flag = false;  public void setFlag() {  flag = true;  }  public boolean getFlag() {  return flag;  }  } |

* In the above example, the setFlag() method sets the flag to true.
* Since the flag is volatile, any changes to its value are immediately visible to other threads.
* Similarly, the getFlag() method returns the value of the flag.

**Synchronized Variables:**

* Synchronized variables are used to provide mutual exclusion between threads.
* When a variable is declared synchronized, only one thread at a time can access the variable.
* Other threads must wait until the thread that currently holds the lock on the variable releases it.
* This ensures that the variable’s value is accessed by only one thread at a time, thus avoiding race conditions.
* For example, let’s say we have a shared list that multiple threads can modify concurrently.
* If we use a synchronized list, we can ensure that only one thread at a time can modify the list, like this:

|  |
| --- |
| import java.util.Collections;  import java.util.List;  import java.util.ArrayList;  public class SynchronizedList {  private List<Integer> list = Collections.synchronizedList(new ArrayList<Integer>());  public void addToList(Integer number) {  list.add(number);  }  public List<Integer> getList() {  return list;  }  } |

* In the above example, the addToList() method adds an integer to the list.
* Since the list is synchronized, only one thread at a time can modify the list.
* Similarly, the getList() method returns the value of the list.

**Atomic Variables Example Code:**

|  |
| --- |
| import java.util.concurrent.atomic.AtomicInteger;  public class AtomicCounterExample {  private static AtomicInteger counter = new AtomicInteger(0);  public static void main(String[] args) {  // create and start multiple threads  for (int i = 0; i < 10; i++) {  new Thread(() -> {  // increment the counter  int value = counter.incrementAndGet();  System.out.println("Counter value: " + value);  }).start();  }  }  }  Counter value: 1  Counter value: 10  Counter value: 9  Counter value: 5  Counter value: 8  Counter value: 7  Counter value: 6  Counter value: 4  Counter value: 2  Counter value: 3 |

* In this example, we create an AtomicInteger called counter with an initial value of 0. We then create 10 threads and start each thread using a lambda expression.
* Inside the lambda expression, we use the incrementAndGet() method provided by the AtomicInteger class to increment the value of the counter atomically. We also print the value of the counter to the console for each thread.
* Because we are using an atomic variable, we can safely increment the counter from multiple threads without worrying about race conditions. The incrementAndGet() method guarantees that the value of the counter is incremented atomically and that the new value is immediately visible to other threads.
* When we run this code, we should see the counter value increase by 1 for each thread, resulting in a total counter value of 10.

**Volatile Example Code:**

|  |
| --- |
| public class VolatileExample {  private static volatile boolean flag = false;  public static void main(String[] args) {  // create and start a new thread  new Thread(() -> {  while (!flag) {  // do some work  }  System.out.println("Thread finished");  }).start();  // set the flag to true after a delay  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  flag = true;  }  } |

* Thread finished
* In this example, we create a volatile boolean variable called flag and set its initial value to false. We then create a new thread using a lambda expression that runs in a loop until the flag is set to true. Inside the loop, we perform some work (which we've omitted for brevity).
* In the main thread, we set the flag variable to true after a delay of 1 second using the Thread.sleep() method. This change to the flag variable should be immediately visible to the other thread because we are using a volatile variable.
* When we run this code, we should see the message “Thread finished” printed to the console after a delay of 1 second, indicating that the other thread has exited the loop and finished its work. This demonstrates the use of a volatile variable to ensure immediate visibility of changes across multiple threads.

**Synchonized Example Code:**

|  |
| --- |
| public class SynchronizedExample {  private static int counter = 0;  public static void main(String[] args) {  // create and start multiple threads  for (int i = 0; i < 10; i++) {  new Thread(() -> {  synchronized (SynchronizedExample.class) {  // increment the counter  counter++;  System.out.println("Counter value: " + counter);  }  }).start();  }  }  }  Counter value: 1  Counter value: 2  Counter value: 3  Counter value: 4  Counter value: 5  Counter value: 6  Counter value: 7  Counter value: 8  Counter value: 9  Counter value: 10  In this example, we create a static counter variable initialized to 0. We then create 10 threads using a loop and start each thread using a lambda expression. |

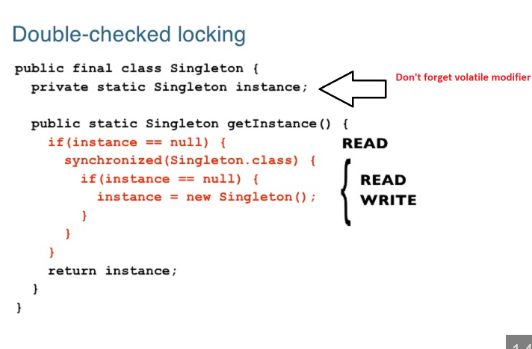
* Inside the lambda expression, we use the synchronized keyword to ensure that only one thread can execute the code block at a time. We synchronize on the SynchronizedExample.class object to ensure that only one thread can access and modify the counter variable at a time.
* Inside the synchronized block, we increment the counter variable and print its value to the console.
* When we run this code, we should see the counter value increase by 1 for each thread, resulting in a total counter value of 10. The use of synchronized variables ensures that multiple threads cannot access and modify the counter variable simultaneously, which prevents race conditions and ensures thread safety.

**Volatile vs. AtomicReference**

* volatile and AtomicReference are two different mechanisms in Java that are used for thread-safe programming. Both ensure the consistency of values in variables that are shared by multiple threads.

**Here are the differences:**

* The volatile keyword guarantees that a variable is stored in a memory area shared by multiple threads. This ensures that different threads see the same value of the variable. However, volatile only makes read/write operations atomic, which means that one thread can change a volatile variable while another thread is reading it. Therefore, volatile variables are only used in simple scenarios where read/write operations need to be atomic.
* The AtomicReference class, on the other hand, ensures atomic reading and writing of a reference variable. The methods of this class automatically perform synchronization operations during reading, updating, or modifying the value of the reference. This makes the reading and updating of a variable thread-safe. The AtomicReference class is used in more advanced scenarios compared to volatile.
* In summary, volatile only makes read/write operations atomic, while AtomicReference ensures thread-safe reading and updating of a reference variable. AtomicReference is used in more advanced scenarios, while volatile is preferred in simpler scenarios.
* In a multi-threaded system this code may not work correctly. **If some other thread accesses the variable and changes it to false, the thread running this code block may not know about the change immediately.** This may result in the loop being executed more than necessary and may result in data integrity problems and it is a hard very hard error to find and fix. A solution to this problem may be to declare the printer\_running variable as volatile. This will make all the changes to the variable by one thread be visible to all other threads at once



**Conclusion**

* In conclusion, atomic, volatile, and synchronized variables are used for managing concurrent access to shared resources in Java. While all three variables are used for the same purpose, they work differently and should beused depending on the specific use case.
* Atomic variables are used when atomic operations on primitive data types are required, such as incrementing or decrementing a counter. Volatile variables are used when multiple threads need to access and modify a variable’s value concurrently, and immediate visibility of changes is required. Synchronized variables are used when mutual exclusion is required to prevent multiple threads from accessing and modifying a shared resource simultaneously.
* It’s important to note that the use of synchronized variables can result in performance issues if used incorrectly. Synchronization can cause contention between threads and can lead to thread blocking and waiting, which can significantly affect the performance of an application. In some cases, using non-blocking data structures and techniques such as locks and semaphores may be a better alternative to using synchronized variables.
* In summary, atomic, volatile, and synchronized variables are essential tools in Java for managing concurrent access to shared resources. Understanding their differences and use cases can help developers write efficient and thread-safe code.

**Important points about Volatile, Atomicity and Synchronization:**

1. **Volatile** –If we want to share the value across multiple threads and updated values will access from main memory instead of CPU cache Thread local memory. We will get updated value for each and evry thread across the programs.
2. **Atomicity** – We have atomic variables like Atomic Integer, Atomic Booleans, Atomic Double, etc. Every operation should be atomic in nature **either do or die**, complete the operation completely commit or rollback completely is called as transaction. Atomic variables are used to perform atomic operations on primitive data types such as int, long, double, etc. They provide a way to modify the value of the variable atomically (i.e., in one atomic operation), thus avoiding race conditions. This means that an atomic variable operation will complete before another operation can start. In Java, the java.util.concurrent.atomic package provides atomic variables.
3. **Synchronization** – To share the values across multiple threads by using class or object level locking using synchronization blocks or synchronization methods. It will make sure data will be shared in all the places even if multiple threads are updating the variables.

What is the difference between volatile and AtomicInteger in Java concurrency?

Volatile only ensures that the access is atomically, **while Atomics ensure that the operation is atomically.** Atomics like **AtomicInteger ensure**, that such operations happen atomically.

atomicity wants to achieve either transaction should be completely successful or all.

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 locks**, java provides a built in mechanism for locking, the synchronized block. Every object can implicitly act as a lock for purposes of synchronization; these built-in locks are called intrinsic locks.What is interesting with the term intrinsic is that the ownership of a lock is per thread and not per method invocation. That means that only one thread can hold the lock at a given time. What you might also find interesting is the term reentrancy, which allows the same thread to acquire the same lock again. Intrinsic locks are reentrant.
* **Client side locking**, if I understand what you mean, is something different. When you don't have a thread safe class, your clients need to take care about this. They need to hold locks so they can make sure that there are not any race conditions.
* **Extrinsic locking** is, instead of using the built in mechanism of synchronized block which gives you implicit locks to specifically use explicit locks. It is kind of more sophisticate way of locking. There are many advantages (for example you can set priorities). A good starting point is the java documentation about locks

**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 extends Thread {  public synchronized void inner() {  // Code to make synchronized or atomic  }  public synchronized void outer() {  // Code to make synchronized or atomic  inner();  }  @override  public void run() {  outer();  }  }  Reentrancy r = new Reentrancy();  Reentrancy r1 = new Reentrancy();  Thread ta = new Thread(r);  Thread tb= new Thread(r);  Ta.start(); --- lock acquired.  Tb.start();  //outer() – lock acquire, 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.

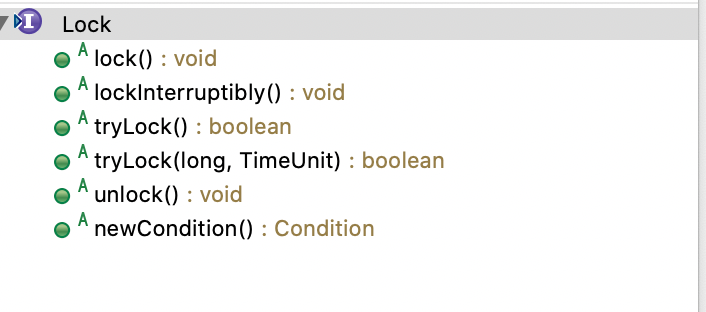
**Extrinsic locks in JAVA**

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) THE CONCEPT OF NULL IN JAVA and how It working with instanceOf operator?

null is a reserved word (keyword) in Java for literal values. It is a literal similar to the true and false. In Java, null is a keyword much like the other keywords public, static or final. It is just a value that shows that the object is referring to nothing. The invention of the word “null” originated to denote the absence of something. For example, the absence of the user, a resource, or anything. But, over the years it puts Java programmers in trouble due to the disturbing null pointer exception. When you declare a boolean variable, it gets its default value as false.

Generally speaking, null are used as a special value to signify:

* Uninitialized state
* Termination condition
* Non-existing object
* An unknown value

**Types of variables in Java**

There are two types of variables in Java. These are reference types and primitive types. Primitive types are the ones that would hold values and reference types are the ones that can store references. Examples of primitive variables include int, Boolean, char, short, float, long and double. The reference variables will store null if they are explicitly referencing an object in memory. The main difference between primitive and reference type is that primitive type always has a value, it can never be null but reference type can be null, which denotes the absence of value.

It is a very important concept in Java. The original intention of inventing null was to denote the absence of something. null has a number of properties as we see below;

Properties of null

null as a default

The reserved word null is case sensitive and cannot be written as Null or NULL as the compiler will not recognize them and will certainly give an error.

Every primitive type of variable has a default value (e.g int has 0, boolean has false) if not initialised at the same time of declaration. Similarly null is the default value of any reference type of variable which is not initialised at the time of declaration. This is true for all kinds of variables, instance variable or static variable, except that the compiler will warn you if you use a local variable without initialising them.

**null is used for casting to other types**

null is neither an object nor a type. It’s just a special value, which can be assigned to any reference type. Typecasting null to any reference type is fine at both compile-time and runtime and it will not throw any error or exception,

It is imperative to note that null can only be assigned to reference types. We cannot assign null to primitive variables e.g int, double, float, or boolean. If we try to do so, then the compiler will complain.

null as an instanceOf operator

**The java instanceof operator which is also known as type comparison operator, tests whether the object is an instance of the specified type (class or subclass or interface). It also compares the instance with type and returns either true or false. If the instanceof operator is applied with any variable that has a null value, it returns false.**

**NullPointer Exception (NPE)**

NullPointerException is a runtime exception. In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when an application attempts to use an object with a null value.

**Autoboxing and unboxing**

During auto-boxing and unboxing operations, the compiler simply throws a Nullpointer exception error if a null value is assigned to a primitive boxed data type.

**Static vs Non static Methods**

We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call a static method with reference variables with null values. Since static methods are bonded using static binding, they won’t throw Null pointer Exception.

**Null Operations**

Operations involving null are extremely fast and easy to perform at run-time. There are only two kinds of operations;

Initialise or set a reference to null (e.g name = null): The only thing to do is to change the content of one memory cell (e.g. setting it 0).

Check if a reference points to null (e.g. if name == null): The only thing to do is to check if the memory cell of the reference holds the value 0.

**Tips to Handle Null**

After elaborating what null is and what it's all about, it would be great to close the discussion with some tips on how you as a developer can handle this issue. Below are some tips you can start using today.

**Use exceptions Over Nulls**

One strange case when you might see people using null is exceptional situations. This is an inherently error prone practice, as critical errors can be omitted or resurface in different places of the system, causing debugging to be a pain. Therefore, always throw an exception instead of returning null if something went wrong.

**Test your code**

Well this advice is related to all kinds of bugs, not just unexpected nulls. Testing your code thoroughly using an environment similar to production is a great way to prevent NPEs. Never release a piece of code without making sure it works. There’s no such thing as “a quick, simple fix that doesn’t require testing.”

**Validate Public API Arguments**

It is possible to use the principle of not passing nulls with reasonable success, but when you’re exposing a public API, you have no control over its users and what they pass to your functions. For this reason, always check the arguments being passed to your public APIs for correctness. If your only concern is the nullity of the argument, consider using the requireNonNull function from the Objects class;

**Use Objects Methods as Stream Predicts**

Although Objects.isNull and Objects.nonNull are not the best fit for typical null checks, they are a perfect fit to use with streams. The filtering or matching lines with them read (arguably) much better than those with operators. This is the very reason they were introduced in the JDK.

**Optional is not for Fields**

Optional was designed to indicate missing return values. One tempting case that it was not designed for and surely is not necessary for is class fields. By means of encapsulation, you should have full control over the field's value, including null. On the other hand, making fields explicitly Optional might bring you weird problems like:

**How should you write a constructor or setter for such a field?**

You have to deal with the Optional even in cases when you’re sure that the value is there.

**How should automapper handle those fields?**

Therefore, use direct references for fields and carefully analyse whether a field can be null or not at any given point. If your class is well-encapsulated, this should be fairly easy.

**Final thoughts**

Null, though can be a bit problematic for both amateur and professional developers, knowing its properties and how to handle it will be a great way to make it less of a barrier to you. Using exceptions over nulls, testing code, validating public API arguments are just some of the tips you can use in dealing with null.

Reference Link: <https://logit.io/blog/post/null-in-java/>

Q) Java Inheritance? – 50%

Q) Why “String” is immutable? – 80%

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

Q) Overloading vs overriding? In terms of exception handling? -- 50%

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

- Map internal working (Java8 improvement)

- Why Generic and how to use?

Q) Volatile, Atomic Integer, Countdownlatch, cyclic barrier and ThreadLocal advantages?

Q) How CPU Performs for Java Execution - <https://medium.com/javarevisited/how-cpu-performs-for-java-execution-9758c771d8c3>

Q) Serialization vs Externalization & Serializations id usage and why?

Q) Race condition problem /producer & consumer problem in Threads?

A race condition in Java emerges when two or more threads concurrently access shared data, and the final outcome hinges on the timing or order of their execution. This can result in unpredictable behavior and elusive bugs, posing challenges during reproduction and debugging.