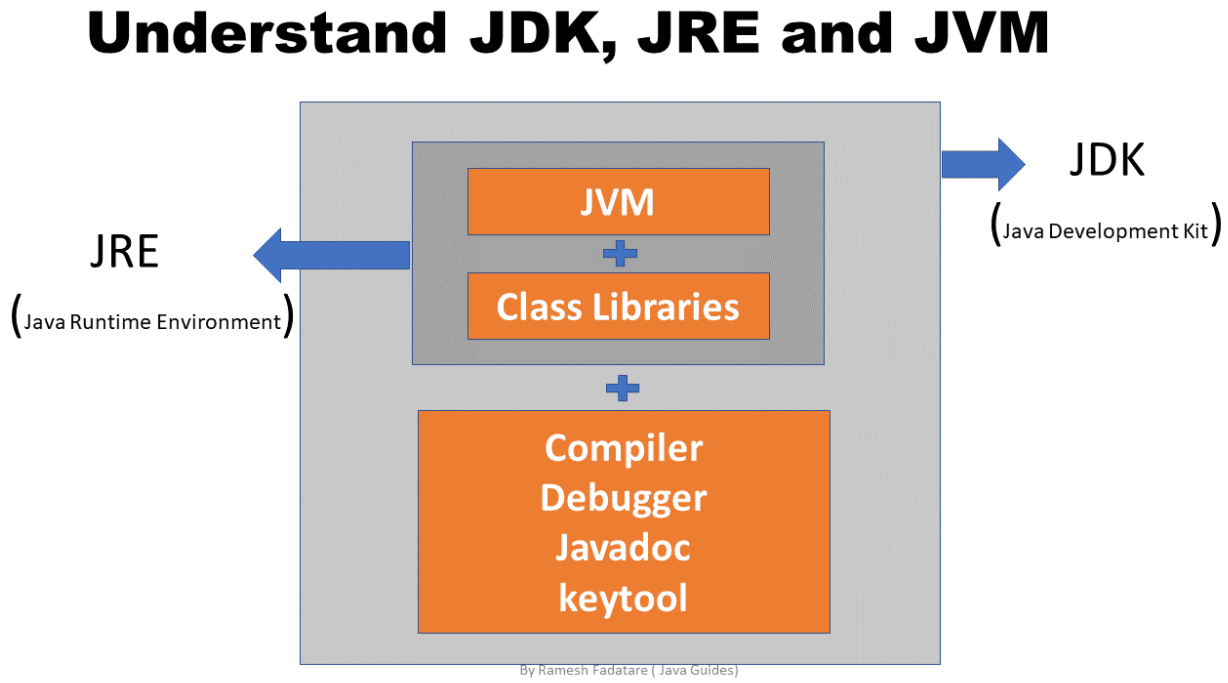
JDK,JRE & JVM

1.JDK(Java development Kit)

The JDK(Java Development Kit) is a superset of the JRE and contains everything that is in the JRE, plus tools such as the compiler, debugger, JavaDoc, keytool etc necessary for developing and running Java programs or applications.



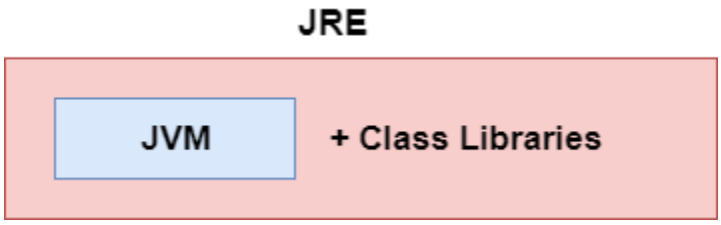
2. JRE(Java Runtime Environment)

The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applications written in the Java programming language.

JRE doesn't contain any development tools such as Java compiler, debugger, JShell, etc.

If you just want to execute a java program, you can install only JRE. You don't need JDK because there is no development or compilation of java source code required.

The below diagram shows the JRE (Java Runtime Environment) is a software package that provides Java class libraries, along with Java Virtual Machine (JVM), and other components to run applications written in Java programming.

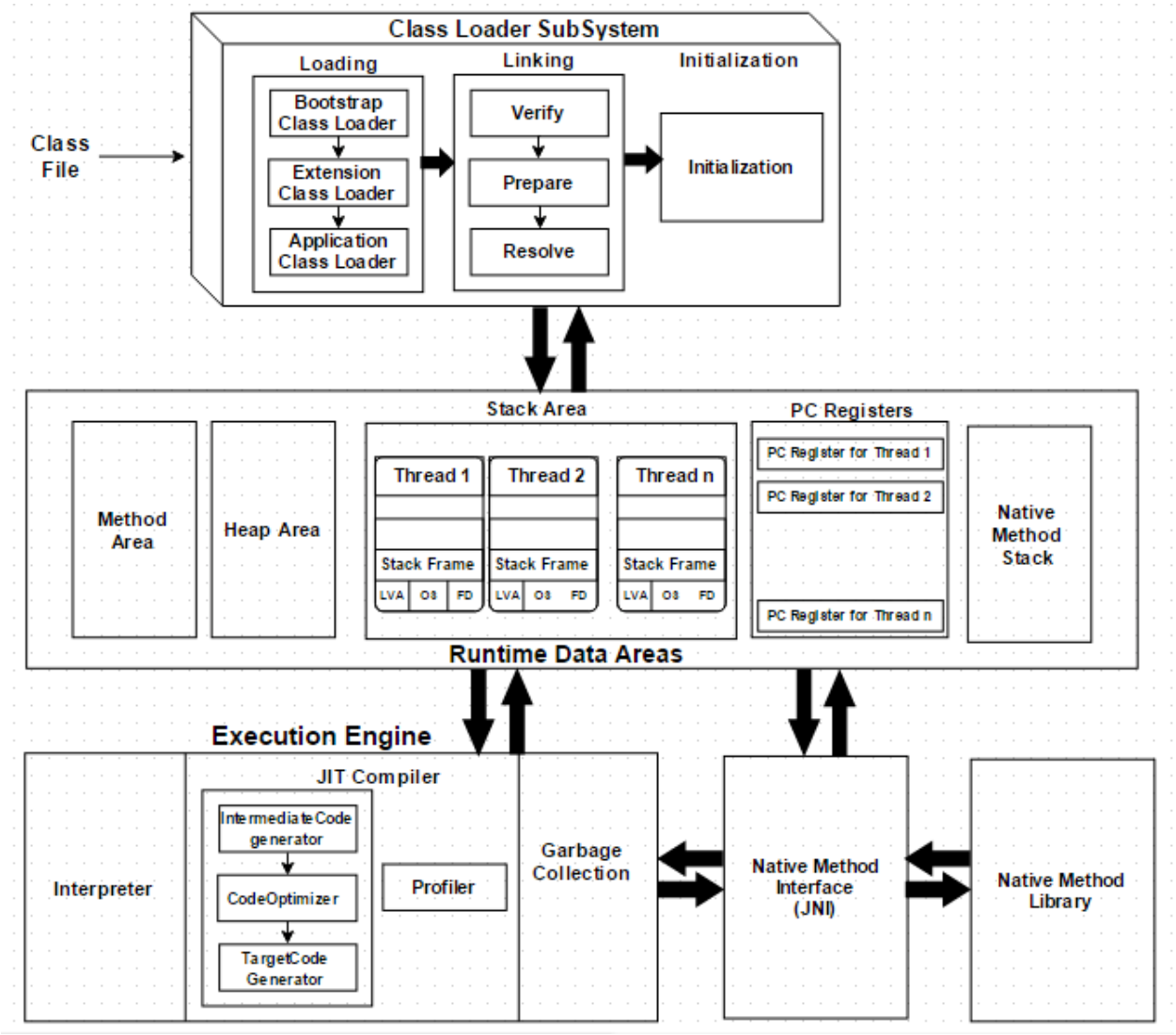


JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc) + runtime libraries.

3. JVM (Java Virtual Machine)

What Is the JVM?

A **Virtual Machine** is a software implementation of a physical machine. Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**. The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file. Below is a diagram of the Architecture of the JVM.



How Does the JVM Work?

As shown in the above architecture diagram, the JVM is divided into three main subsystems:

1. ClassLoader Subsystem
2. Runtime Data Area
3. Execution Engine

What do the Class Loaders do?

When we compile a source (.java) file, it gets converted into byte code as a .class file. Furthermore, when we try to use this class in our program, the class loader loads it into the main memory. Normally the class that contains the main() method is the first class to be loaded into the memory. The class loading process happens in three phases.

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

Phases Of Class Loading

1) Loading

Loading phase involves accepting the binary representation (bytecode) of a class or interface with a specific name, and generating the original class or interface from that. The JVM uses the ClassLoader.loadClass() method for loading the class into memory.

2) Linking

Once a class is loaded into the memory, it undergoes the linking process. Linking a class or interface involves combining the different elements and dependencies of the program together. As the name suggests making required links with each other.

3) Initialization

This is the final stage of class loading. Initialization involves executing the initialization code of the class or interface. This can include calling the class's

constructor, executing the static block, and assigning values to all the static variables.

Class Loaders in Java

The class loaders load the compiled class file. Class loader system contains 3 types of class loaders.
1. Bootstrap class Loader (BCL)
2. Extension class Loader (ECL)
3. Application class Loader (ACL)

Bootstrap class loader

Bootstrap class Loader is responsible for loading all core java API classes. These are all classes that exist inside rt.jar which are available in all JVMs by default. Bootstrap class Loader loads classes from bootstrap class path. However it's implementation is in native languages(C, C++) but not in java. Bootstrap Class Path is JDK/JRE/lib

Extension class loader

Extension Class Loader is a child class of Bootstrap class loader which is responsible for loading all classes from the extension class path in java. However, it's implementation is in java only. The Extension Class Path is : JDK/JRE/lib/ext

Application class loader

Application Class Loader is a child class of Extension class loader which is responsible for loading classes from application class-path. It's implementation is also in java. The Application class path is our environment class path.

How does a class loader work?

Class loaders follow Delegation Hierarchy Principle. When JVM comes across a particular class, first of all it checks whether the corresponding .class file is already loaded or not. If it is already loaded in the Method area, then the JVM considers that class is loaded without fail. But If it is not loaded, JVM requests a class loader subsystem to load that particular class accordingly.

The class loader subsystem hands over the request to the application class loader. Further Application class loader delegates the request to extension class loader which in turn delegates request to bootstrap class loader. Now the Bootstrap Class Loader will search in the bootstrap class path, if it is available then the corresponding .class will be loaded by Bootstrap Class Loader. If it is not available then Bootstrap Class Loader delegates the request to extension class loader.

Further, Extension Class Loader will search in extension class path. if it is available then it will be loaded otherwise Extension Class Loader delegates the request to Application Class Loader subsequently. Now the Application Class Loader will search in the application class path. If it is available, then it will be loaded otherwise in the end we will get a runtime exception saying NoClassDefFoundError or ClassNotFoundException.

*** Loader Priority BCL>ECL>ACL

For example, Methods to get class loaders are as below :

String.class.getClassLoader();

Test.class.getClassLoader();

Customer.class.getClassLoader();

***CustomizedClassLoader extends java.lang.ClassLoader : while developing web server/application server we can use it to customize class loading mechanism.

JVM Memory Area

The division of total Memory area of JVM is in 5 parts :

1. Method Area :

In the method area, One area will be allocated for each JVM. It will be created at JVM startup. Class level binary information & static variables reside in this area. Also Constant pools will be saved inside the method area. Further It can be accessed by multiple threads simultaneously, therefore it is not thread-safe.

2. Heap Area :

One area will be allocated for each JVM. It will be created at JVM startup. Objects reside in this area. It can be accessed by multiple threads simultaneously, therefore it is also not thread-safe.

How can we find allocated heap area ?

```
Runtime r = Runtime.getRuntime();
```

```
r.maxMemory(); r.initialMemory(); r.freeMemory();
```

Runtime is inside the java.lang package is a Singleton class.

How to set maximum & minimum heap sizes ?

1. By using command prompt execution of program :

java -Xmx512m JavaProgramFileName enter
java -Xms64m JavaProgramFileName enter
Where Xmx indicates Maximum Memory & Xms indicates Minimum Memory. Heap memory is finite memory but based on our requirement we can set max & min heap sizes.

2. By Setting 'JAVA_OPTS' as a system variable

JAVA_OPTS="-Xms256m -Xmx512m"
After that in a command prompt run the following command:
SET JAVA_OPTS="-Xms256m -Xmx512m"
This setting indicates
allocating minimum 256MBs of heap
allocating maximum 512MBs of heap

3. Stack Area :

It is available per thread unlike Method & Heap area as they are one per JVM. Each entry in stack is called Stack frame or activation record. Also it is thread-safe as it allocates one memory for each thread. Furthermore, Each stack frame has three parts : local variable array, operand stack and frame data.
Local Variable Array : It contains values of local variables & method parameters.
Operand Stack : JVM uses it as workspace, some instructions push the values to it & some pop from it & some other to perform arithmetic operations.
Frame Data : It contains all symbolic references related to the method. It also contains reference of exceptions related to the method.

4. PC Registers :

(Program Counter Registers) : Internally used by JVM. For every thread JVM creates a separate PC register. In brief, the PC register contains the address of currently executing threads.

5. Native Method Stacks :

For every thread JVM creates a separate native method stack if its native method calls.

****Note :** Method Area, Heap Area, Stack Area are also considered as important memory areas from a programmers point of view. Method Area, Heap area are for per JVM whereas Stack area, PC register & the native method stack are for per thread.

♦ **Static Variables are stored in Method area**

♦ **Instance Variables are stored in Heap area**

♦ **Local Variables are stored in Stack area**

Execution Engine

The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

1. **Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
2. **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    1. **Intermediate Code Generator** – Produces intermediate code
    2. **Code Optimizer** – Responsible for optimizing the intermediate code generated above
    3. **Target Code Generator** – Responsible for Generating Machine Code or Native Code
    4. **Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

The primary purpose of the JIT compiler is to improve performance. In fact, Internally it maintains a separate count for every method. Whenever a JVM comes across any method call, first that method is interpreted normally by the interpreter and the JIT compiler increments the corresponding count variable accordingly.

This process continues for every method. Once any method count reaches threshold value then the JIT compiler identifies that the method is a repeatedly used method. We also call that method a hotspot for the JIT

compiler. Then JIT Compiler compiles that method immediately & generates corresponding native code. Next time JVM comes across that method call, then JVM uses native code directly & executes it instead of interpreting it once again so that performance of the system will be improved. However the threshold count varies from JVM to JVM.

♦ Some advanced JIT compilers recompile generated native code if count reached threshold value second time so that more optimized machine code can be generated. Internally profiler (which is the part of JIT compiler) is responsible to identify hot-spots.

****Note : =>** JVM interprets total program at-least once.
However, JIT compilation is applicable only for repeatedly required methods not for every method.
Additionally JIT Compiler has intermediate code generator, code optimizer, target code generator & machine code generator in its whole compilation process.

3. **Garbage Collector**: Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

Java Native Interface (JNI): JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

Native Method Libraries: This is a collection of the Native Libraries, which is required for the Execution Engine.

What is difference between PermGen & Metaspace?

PermGen (JDK 7 & lower versions)

PermGen is a special heap space separated from the main memory heap. Moreover, the full form of the PermGen is the Permanent Generation. JVM uses this space to keep track of metadata such as loaded classes. Additionally, the JVM stores all the static content in this space such as all the static methods, primitive variables, and references to the static objects.

The default maximum memory size for 32-bit JVM is 64 MB and for the 64-bit version its 82 MB. However, we can change the default size with the JVM options. When application tries to load unusual number of classes, we may see **"java.lang.OutOfMemoryError : PermGen space"**

Most importantly, **Oracle has completely removed this memory space in the JDK 8 release.**

Metaspace (JDK 8 & higher versions)

Metaspace is a new memory space – starting from the Java 8 version. Specifically, it has completely replaced the older PermGen memory space. The most significant difference is how it handles memory allocation. It is a part of native memory region. Moreover, Metaspace by default auto increases its size depending on the underlying OS. Here, the garbage collection is automatically triggered when the class metadata usage reaches its maximum metaspace size. Moreover, it does better garbage collection than PermGen. Hence, with this improvement, JVM reduces the chance to get the java.lang.OutOfMemor*y* error.

The PermGen is garbage collected like the other parts of the heap. The thing to note here is that the PermGen contains meta-data of the classes and the objects i.e. pointers into the rest of the heap where the objects are allocated.

Metaspace garbage collection is triggered when the class metadata usage reaches MaxMetaspaceSize limit.

From earlier Java Versions till Java 6, the Heap is being divided into the young and old generations.
So in the Java 6 world, then there is a further part of the Heap called PermGen. This stands for the permanent generation. Objects which are in the PermGen will survive forever. The PermGen is never garbage-collected. This means that if the PermGen runs out of space, then your application will crash. There are two types of object that go into PermGen:

1. internalized strings that is strings which are placed into a pool for reuse.
2. every time we create a class, the meta data for that class is placed into PermGen.

In computer science, string interning is a method of storing only one copy of each distinct string value, which must be immutable.

For each class in our application, there is some memory needed to store information about the class and that lives in the PermGen space.
Now you may have come across problems in the past where your application crashes with an error message saying that it's ran out of PermGen space. This

is an annoying error because when the PermGen runs out of space, it simply means that your application has too many classes or internalized strings. Unlike other out of memory errors, it's not caused by memory leaks or faults in your code. The only thing you can to do avoid PermGen error is to increase the size of the allocated memory for PermGen within the Heap.
If you're running a server application for example, you're using a website using Tomcat, that each time you re-deploy your application, if you make a small change to your code and you deploy that version to the server, even though many of the classes won't have changed, a new complete set of meta data for all your classes is created in the PermGen; and all the meta data relating to the previous deployments of your application is still in the PermGen but will never be referred to. Because there's no garbage collection on the PermGen, it will never get cleared out. And what this means is that if you re-deploy onto a large server a number of times, eventually the PermGen will run out of space. For this reason, on a live server, you might want to consider stopping and restarting Tomcat or whatever your service application is each time you deploy a new version of the code to avoid the PermGen becoming full. So that was how the Heap worked under Java 6 and earlier versions of Java.
From Java 7, internalized strings are no longer stored in the PermGen. This change means that internalized strings are in the old part of the Heap, and can therefore be garbage collected. If your application uses lots of strings, then this won't be a contributing factor to PermGen crashes anymore.
From Java 8, well in Java 8 they remove the PermGen altogether. Instead they created something else called the MetaSpace, which is where the meta data field classes are placed. The MetaSpace is a separate area of memory and it's not part of the Heap. Instead it's allocated out of your computer's native memory so the maximum available space for the MetaSpace is the total available system memory for your computer. Now there is an option to cap the maximum size for the MetaSpace; if you don't do that then the virtual machine will just grow the MetaSpace as it needs to. But a useful feature of the MetaSpace is that when classes are no longer creatable, the meta data

related to those classes is then removed. So this means that if you're running Tomcat for example, under Java 8, then every time you re-deploy, all the old class meta data in the MetaSpace will get removed, and the MetaSpace won't grow and grow each time you re-deploy.

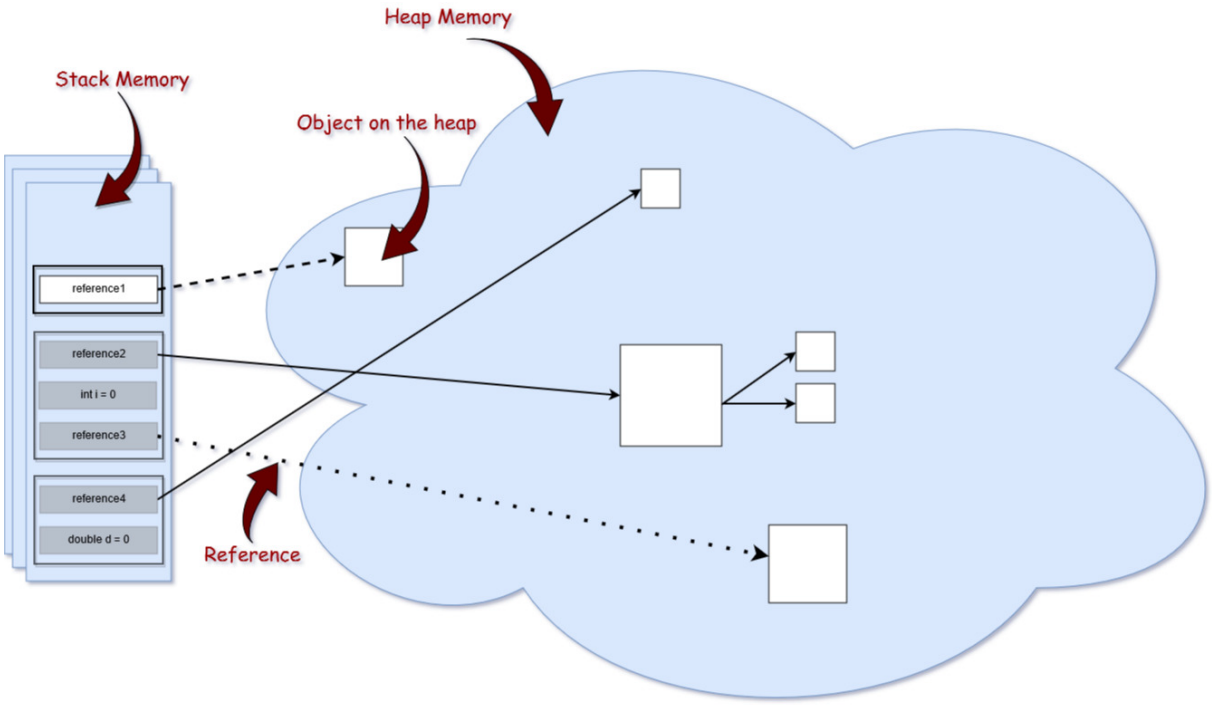
Garbage collection

You might think that if you are programming in Java, what do you need to know about how memory works? Java has automatic memory management, a nice and quiet garbage collector that works in the background to clean up the unused objects and free up some memory.

Therefore, you as a Java programmer do not need to bother yourself with problems like destroying objects, as they are not used anymore. However, even if this process is automatic in Java, it does not guarantee anything. By not knowing how the garbage collector and Java memory is designed, you could have objects that are not eligible for garbage collecting, even if you are no longer using them.

So knowing how memory actually works in Java is important, as it gives you the advantage of writing high-performance and optimized applications that will never ever crash with an **OutOfMemoryError**. On the other hand, when you find yourself in a bad situation, you will be able to quickly find the memory leak.

To start with, let’s have a look at how the memory is generally organized in Java:



The Stack

Stack memory is responsible for holding references to heap objects and for storing value types (also known in Java as primitive types), which hold the value itself rather than a reference to an object from the heap.

In addition, variables on the stack have a certain visibility, also called **scope**. Only objects from the active scope are used. For example, assuming that we do not have any global scope variables (fields), and only local variables, if the compiler executes a method's body, it can access only objects from the stack that are within the method's body. It cannot access other local variables, as those are out of scope. Once the method completes and returns, the top of the stack pops out, and the active scope changes.

Maybe you noticed that in the picture above, there are multiple stack memories displayed. This is due to the fact that the stack memory in Java is allocated per Thread. Therefore, each time a Thread is created and started, it

has its own stack memory — and cannot access another thread's stack memory.

The Heap

This part of memory stores the actual object in memory. Those are referenced by the variables from the stack. For example, let's analyze what happens in the following line of code:

```
StringBuilder builder = new StringBuilder();
```

The new keyword is responsible for ensuring that there is enough free space on heap, creating an object of the StringBuilder type in memory and referring to it via the "builder" reference, which goes on the stack.

There exists only one heap memory for each running JVM process. Therefore, this is a shared part of memory regardless of how many threads are running. Actually, the heap structure is a bit different than it is shown in the picture above. The heap itself is divided into a few parts, which facilitates the process of garbage collection.

The maximum stack and the heap sizes are not predefined — this depends on the running machine. However, later in this article, we will look into some JVM configurations that will allow us to specify their size explicitly for a running application.

There are three steps involved in Garbage Collection:

Mark

In this stage, GC (Garbage collector) will scan the heap memory segment and mark all of the live objects which are having references by the application. All of the objects which are not having any references are eligible for garbage collection.

Sweep

In the sweep stage, GC will recycle all the unreferenced objects from the heap memory.

Compact

After sweep, there are many regions in the heap memory that become empty which causes fragmentation. The compact phase helps in arranging the objects in the contiguous blocks at the start of the heap. This will help in allocating the new objects in sequence.

Heap Memory and Its Partition

Java objects created by the application reside in the memory called heap memory. Heap is created when JVM starts, and as per application usability, heap usage increases and gets full. When heap usage becomes full and further request for new object allocation doesn't have any free space in the current heap memory, then garbage collection happens. In the Garbage Collection above, three steps such as Mark, Sweep, and Compaction happen, which removes all the unreferenced objects from the heap memory and creates room for new object allocation.

Before understanding the heap memory segment we need to understand the concept of Generational Garbage Collection in Java.

What Is Generational Garbage Collection in Java?

In GC, marking, sweeping, and compacting are performed for all unreferenced objects from heap memory. As more and more objects allocate, the JVM heap is piled up with a large number of object allocations, which creates a longer time need for garbage collection. However, empirical or hypothetical analysis of applications has shown that most objects are short-lived. Therefore, marking and compacting all the objects from JVM

heap memory is inefficient and time-consuming. For this reason, GC implements a generational garbage collection that categorizes objects based on their age (lifespan). With this process, objects are allocated in different generations and are garbage collected accordingly.

Heap memory is divided into majorly 2 areas:

1. Young generation(Nursery space)

2. Old generation

1. Young Generation (Nursery Space)

Whenever new objects are created they are allocated to the young generation. The young generation basically consist of two partitions.

A) Eden Space

All of the new objects are first allocated in the Eden space.

B) Survivor Space

After one GC cycle, all live objects from Eden space are moved to Survivor space. Survivor space is further divided into two parts, s1 & s2 space, also called FromSpace and ToSpace. Both Survivor spaces always start empty when JVM starts.

Below is the normal object allocation flow that happens in the young generation:

- At first, all the new objects are allocated into Eden space while both the survivor spaces are empty.

- When Eden space has filled there is no further space for new object allocation, it causes allocation failure and minor GC happens. In this stage of GC, all live objects are marked and moved to S1 space and Eden space is cleared while S2 is still empty.

- Next, when another minor GC happens, it will mark all the live objects from Eden & S1 space. With GC it clears all the live objects from Eden and S1 Space and moves them into S2 space. With this, Eden and S1 space will be empty. At any point in time, one of the survivors is always empty.

- In the next minor GC, the same process happens for Eden and S2 space after GC all the live objects are moved into S1.

2. Old Generation

When objects are long-lived in the young generation with multiple GC cycles, they are marked live in the survivor space. They will be eligible for promotion to old generations after completing the threshold of the GC cycle. (By default, modern JVMs threshold is set to 15 GC cycles) These long-lived objects are further moved into the old generation. The old generation is also known as the tenured generation. The garbage collection events in this area are called major collections.

Full GC performs cleaning of all the generations(young + old generations). It performs promotion of all the live objects from the young generation to old generation, as well as compaction of the old generation. Full GC is the stop-the-world pause which will ensure no new objects are allocated and objects do not suddenly become unreachable while GC performs.

Types of Garbage Collection (GC)

The Garbage Collection events cleaning out different parts inside heap memory are often called Minor, Major, and Full GC events. But, as the terms Minor, Major, and Full GC are widely used and without a proper definition, we will have a look at the explanation of all these GC. JVM runs the garbage collector if it senses that memory is running low.

Minor GC

Collecting garbage from the Young Generation space is called Minor GC. Minor GC cleans the Young Generation. Minor GC is always triggered when the JVM is unable to allocate space for a new object, i.e., when the Eden space is getting full. So the higher the allocation rate, the more frequently Minor GC occurs.

Major GC

Major GC is cleaning the Tenured (Old space). As OLD Gen is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a 'major GC' has occurred. The old generation collector will try to predict when it needs to collect to avoid a promotion failure from the young generation.
The collectors track a fill threshold for the old generation and begin collection when this threshold is passed. If this threshold is not sufficient to meet promotion requirements then a 'FullGC' is triggered.

Full GC

Full GC is cleaning the entire Heap — both Young and Old spaces. Many people get confused with Major (Only OLD generation) and Full GC (Young + OLD(Heap)). A FullGC involves promoting all live objects from the young generation to the OLD generation after collection and compaction of the old generation. Full GC will be a Stop-the-World pause. Stop-the-World is making sure that new objects are not allocated and objects do not suddenly become unreachable while the collector is running.

Types of Garbage Collection

There are five types of garbage collection are as follows:

1. Serial Collector

The serial collector uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads.

It is best-suited to single processor machines because it can't take advantage of multiprocessor hardware, although it can be useful on multiprocessors for applications with small data sets. The serial collector is selected by default on certain hardware and operating system configurations or can be explicitly enabled with the option -XX:+UseSerialGC.

Serial GC would be the best choice for applications that do not have low pause requirements and work on very small heap sizes.

2. Parallel Collector

The parallel collector is also known as a throughput collector, and is a generational collector similar to the serial collector. The main difference between the serial and parallel collectors is that the parallel collector has multiple threads that are used to speed up garbage collection.

The parallel collector is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. You can enable it by using the -XX:+UseParallelGC option.

Parallel compaction is a feature that enables the parallel collector to perform major collections in parallel. Without parallel compaction, major collections are performed using a single thread, which can significantly limit scalability.

Parallel compaction is enabled by default if the option -XX:+UseParallelGC has been specified. You can disable it by using the -XX:-UseParallelOldGC option.

Parallel Collector would be the best choice where throughput is more important than latency. One can use Parallel Collector where long pauses are acceptable, such as bulk data processing or batch jobs.

Parallel Old GC: It is similar to parallel GC, except that it uses multiple threads for both generations.

Concurrent Mark Sweep (CMS) Collector: It does the garbage collection for the old generation. You can limit the number of threads in CMS collector using XX:ParalleCMSThreads=JVM option. It is also known as Concurrent Low Pause Collector.

G1 Garbage Collector: It introduced in Java 7. Its objective is to replace the CMS collector. It is a parallel, concurrent, and CMS collector. There is no young and old generation space. It divides the heap into several equal-sized heaps. It first collects the regions with lesser live data.

HashCode and Equals

By default, the Java super class ***java.lang.Object*** provides two important methods for comparing objects: ***equals()*** and ***hashcode()***. These methods become very useful when implementing interactions between several classes in large projects. In this article, we will talk about the relationship between these methods, their default implementations, and the circumstances that force developers to provide a custom implementation for each of them.

Method Definition and Default Implementation

- ***equals(Object obj):*** a method provided by ***java.lang.Object*** that indicates whether some other object passed as an argument is ***"equal to"*** the current instance. The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.
- ***hashcode():*** a method provided by ***java.lang.Object*** that returns an integer representation of the object memory address. By default, this method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

The Contract Between equals() and hashcode()

The default implementation is not enough to satisfy business needs, especially if we're talking about a huge application that considers two objects as equal when some business fact happens. In some business scenarios, developers provide their own implementation in order to force their own equality mechanism regardless of the memory addresses.

As per the Java documentation, developers should override both methods in order to achieve a fully working equality mechanism — it's not enough to just implement the ***equals()*** method.

If two objects are equal according to the equals(Object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.

In the following sections, we provide several examples that show the importance of overriding both methods and the drawbacks of overriding ***equals()*** without ***hashcode()***.

Practical Example

We define a class called ***Student*** as the following:

```java
package com.programmer.gate.beans;

public class Student {

    private int id;
    private String name;

    public Student(int id, String name) {
        this.name = name;
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

For testing purposes, we define a main class ***HashcodeEquals*** that checks whether two instances of ***Student*** (who have the exact same attributes) are considered as equal.

```java
public class HashcodeEquals {

    public static void main(String[] args) {
        Student alex1 = new Student(1, "Alex");
        Student alex2 = new Student(1, "Alex");

        System.out.println("alex1 hashcode = " + alex1.hashCode());
        System.out.println("alex2 hashcode = " + alex2.hashCode());
        System.out.println("Checking equality between alex1 and alex2 = " + alex1.equals(alex2));
    }
}
```

Output:

```
alex1 hashcode = 1852704110
alex2 hashcode = 2032578917
Checking equality between alex1 and alex2 = false
```

Although the two instances have exactly the same attribute values, they are stored in different memory locations. Hence, they are not considered equal as per the default implementation of ***equals()***. The same applies for ***hashcode()*** — a random unique code is generated for each instance.

Overriding equals()

For business purposes, we consider that two students are equal if they have the same *ID*, so we override the ***equals()*** method and provide our own implementation as the following:

```java
@Override
public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof Student))
        return false;
    if (obj == this)
        return true;
    return this.getId() == ((Student) obj).getId();
}
```

In the above implementation, we are saying that two students are equal if and only if they are stored in the same memory address **OR** they have the same ID. Now if we try to run ***HashcodeEquals***,

we will get the following output:

```
alex1 hashcode = 2032578917
alex2 hashcode = 1531485190
Checking equality between alex1 and alex2 = true
```

As you noticed, overriding ***equals()*** with our custom business forces Java to consider the ID attribute when comparing two ***Student*** objects.

equals() With ArrayList

A very popular usage of ***equals()*** is defining an array list of ***Student*** and searching for a particular student inside it. So we modified our testing class in order to achieve this.

```
public class HashcodeEquals {

    public static void main(String[] args) {
        Student alex = new Student(1, "Alex");

        List < Student > studentsLst = new ArrayList < Student > ();
        studentsLst.add(alex);

        System.out.println("Arraylist size = " + studentsLst.size());
        System.out.println("Arraylist contains Alex = " + studentsLst.contains(new Student(1, "Alex")));
    }
}
```

fter running the above test, we get the following output:

```
Arraylist size = 1
Arraylist contains Alex = true
```

Overriding hashcode()

Okay, so we override ***equals()*** and we get the expected behavior — even though the hash code of the two objects are different. So, what's the purpose of overriding ***hashcode()***?

equals() With HashSet

Let's consider a new test scenario. We want to store all the students in a ***HashSet***, so we update ***HashcodeEquals*** as the following:

```
public class HashcodeEquals {

    public static void main(String[] args) {
        Student alex1 = new Student(1, "Alex");
        Student alex2 = new Student(1, "Alex");

        HashSet < Student > students = new HashSet < Student > ();
        students.add(alex1);
        students.add(alex2);

        System.out.println("HashSet size = " + students.size());
        System.out.println("HashSet contains Alex = " + students.contains(new Student(1, "Alex")));
    }
}
```

If we run the above test, we get the following output:

```
HashSet size = 2
HashSet contains Alex = false
```

WAIT!! We already override ***equals()*** and verify that *alex1* and *alex2* are equal, and we all know that ***HashSet*** stores unique objects, so why did it consider them as different objects ?

HashSet stores its elements in memory buckets. Each bucket is linked to a particular hash code. When calling *students.add(alex1)*, Java stores *alex1* inside a bucket and links it to the value of *alex1.hashcode()*. Now any time an element with the same hash code is inserted into the set, it will just replace *alex1*. However, since *alex2* has a different hash code, it will be stored in a separate bucket and will be considered a totally different object.

Now when ***HashSet*** searches for an element inside it, it first generates the element's hash code and looks for a bucket which corresponds to this hash code.

Here comes the importance of overriding ***hashcode()***, so let's override it in ***Student*** and set it to be equal to the ID so that students who have the same ID are stored in the same bucket:

```
@Override
public int hashCode() {
    return id;
}
```

Now if we try to run the same test, we get the following output:

```
HashSet size = 1
HashSet contains Alex = true
```

See the magic of ***hashcode()***! The two elements are now considered as equal and stored in the same memory bucket, so any time you call *contains()* and pass a student object holding the same hash code, the set will be able to find the element.

The same is applied for ***HashMap, HashTable***, or any data structure that uses a hashing mechanism for storing elements.

Conclusion

In order to achieve a fully working custom equality mechanism, it is mandatory to override ***hashcode()*** each time you override ***equals().*** Follow the tips below and you'll never have leaks in your custom equality mechanism:

- If two objects are equal, they MUST have the same hash code.
- If two objects have the same hash code, it doesn't mean that they are equal.
- Overriding ***equals()*** alone will make your business fail with hashing data structures like: ***HashSet, HashMap, HashTable*** ... etc.
- Overriding ***hashcode()*** alone doesn't force Java to ignore memory addresses when comparing two objects.