Java Fundamentals

for

Architects & Developers

Neeraj Sachan

Preface –

Over the past 18 years, I have had the privilege of working deeply with Java and the Java Virtual Machine (JVM), traversing the evolution of this powerful ecosystem — from monolithic applications to distributed microservices, from basic servlets to reactive systems, and from XML-based configurations to modern annotation-driven frameworks. This journey, shaped by practical challenges and architectural decisions, forms the book for important concepts in java.

The intent behind writing this book is to bridge the gap between theoretical concepts and real-world application architecture. While Java as a language continues to evolve, the underlying principles of designing robust, scalable, and maintainable systems remain constant. Through countless production deployments, performance bottlenecks, architectural redesigns, and framework migrations, I have distilled a set of core practices, tools, and patterns that every aspiring or practicing technical architect should internalize.

This book is structured to provide both breadth and depth — covering the internals of the JVM, performance tuning, modularization, memory management, as well as higher-level design considerations like service decomposition, polymorphism, Aggregation association etc. It also explores modern Java features and their impact on system design, drawing parallels between language constructs and architectural outcomes.

Whether you're a senior developer preparing to step into an architect's role, or an architect seeking clarity on evolving Java paradigms, this book aims to serve as both a reference and a roadmap. My experience spans various domains — finance, gaming, investment, and video domain — and I have seen how the right use of Java and JVM capabilities can dramatically affect system longevity and agility.

I invite you to not just read, but challenge and reflect on the material in this book. Architecture is a living discipline — and it’s shaped as much by constraints as by creativity.

Welcome to a journey through the architecture of Java systems, refined by years of experience, and driven by a passion for building things that last.

Table of Contents

1. Object Oriented programming (OOP)

The object-oriented programming revolves around the concept of Objects. **What is an Object**? An object is an instance of a Class. It contains properties and functions. They are like real-world objects. For example, your car, house, laptop, etc. are all objects. They have some specific properties and methods to perform some action. **What is a Class**? The Class defines the blueprint of Objects. They define the properties and functionalities of the objects. For example, Laptop is a class, and your laptop is an instance of it.

Core OOPS concepts are:

i. Abstraction:

Abstraction is the concept of **hiding the internal details and describing things in simple terms**. For example, a method that adds two integers. The internal processing of the method is hidden from the outer world. There are many ways to achieve abstraction in object-oriented programming, such as encapsulation and inheritance. A Java program is also a great example of abstraction. Here java takes care of converting simple statements to machine language and hides the inner implementation details from the outer world.

ii. Encapsulation:

Encapsulation is the technique used to **implement abstraction** in object-oriented programming. Encapsulation is used for access restriction to class members and methods. Access modifier keywords are used for encapsulation in object-oriented programming. For example, encapsulation in java is achieved using private, protected and public keywords.

iii. Polymorphism:

Polymorphism is the concept where an object behaves differently in different situations. There are **two** types of polymorphism - **compile time** polymorphism and **runtime** polymorphism. Compile-time polymorphism is achieved by method overloading. Compiler will be able to identify the method to invoke at compile-time, hence it's called compile-time polymorphism. Runtime polymorphism is implemented when we have an "IS-A" relationship between objects. This is also called a method overriding because the subclass has to override the superclass method for runtime polymorphism.

iv. Inheritance:

Inheritance is the object-oriented programming concept where an object is based on another object. Inheritance is the mechanism of **code reuse**. The object that is getting inherited is called the superclass and the object that inherits the **superclass** is called a **subclass**. We use ***extends*** keyword in java to implement inheritance. Below is a simple example of inheritance in java.

v. Association:

Association is the OOPS concept to define the relationship between objects. The association defines the multiplicity between objects. For example, Teacher and Student objects. There is a one-to-many relationship between a teacher and students. Similarly, a student can have a one-to-many relationship with teacher objects. However, both student and teacher **objects are independent of each other**.

vi. Aggregation:

Aggregation is a special type of association. In aggregation, **objects have their own life cycle but there is ownership. Whenever** we have "HAS-A" relationship between objects and ownership then it's a case of aggregation.

vii. Composition:

Composition is a special case of aggregation. The composition is a **more restrictive form of aggregation.** When the contained object in "HAS-A" relationship can't exist on its own, then it's a case of composition.

For example, House has-a Room. Here the room can't exist without the house. Composition is said to be better than inheritance.

- *Note: In Composition, one object is OWNER of another object while in Aggregation one object is just USER of another object.*

2. Programming paradigm:

i. **Imperative programming** – defines computation as statements that change a program state.(Java)
ii. **Procedural (structured) programming,**– specifies the steps a program must take to reach a desired state. (C)
iii. **Declarative programming** – defines program logic, but not detailed control flow.(SQL)
iv. **Object-oriented programming (OOP)** – organizes programs as objects: data structures consisting of data fields and methods together with their interactions.
v. **Event-driven programming** – program control flow is determined by events, such as sensor inputs or user actions (mouse clicks, key presses) or messages from other programs or threads.
vi. **Automata-based programming** – a program, or part, is treated as a model of a finite state machine or any other formal automaton.
vii. **Functional Programming –** Functional programming is a programming paradigm in which we **try to bind everything in pure mathematical functions style**. It is a declarative type of programming style. Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

3. JDK, JRE and JVM:

i. **Java Development Kit (JDK)** is a complete software environment for building applications and applets using the Java programming language. It is platform-dependent. Therefore, it has different OS platform versions for Windows, Linux, Mac, etc. It allows reading, writing, and executing the Java program. Includes various tools required for writing Java programs.
ii. **JRE** software includes JVM and class libraries to run Java programs independently. Although it can execute the code. Yet, JRE comes bundled with Java Development Kit (JDK) to provide a complete application development experience.
iii. **Java Virtual Machine** (JVM) is an abstract machine responsible for compiling and executing Java code. It is a part of the Java Runtime Environment (JRE), which calls the main function of a program

JDK = Development Tools + JRE (Java Runtime Environment)

JRE = JVM + Class Libraries (For Running the Java Applications)

- *Note: JVM can't be installed alone. As JVM is a part of JRE, you need to install JRE. JVM comes within it.*

4. Class loader:

The Class Loader subsystem is the foundation of JVM's architecture. The process can be broken down into the following phases:

i. **Loading**:
In this phase, the Class Loader reads the binary data of a class file and brings it into the runtime data area. It does this for classes referenced by the currently executing class, ensuring that classes are loaded as needed.

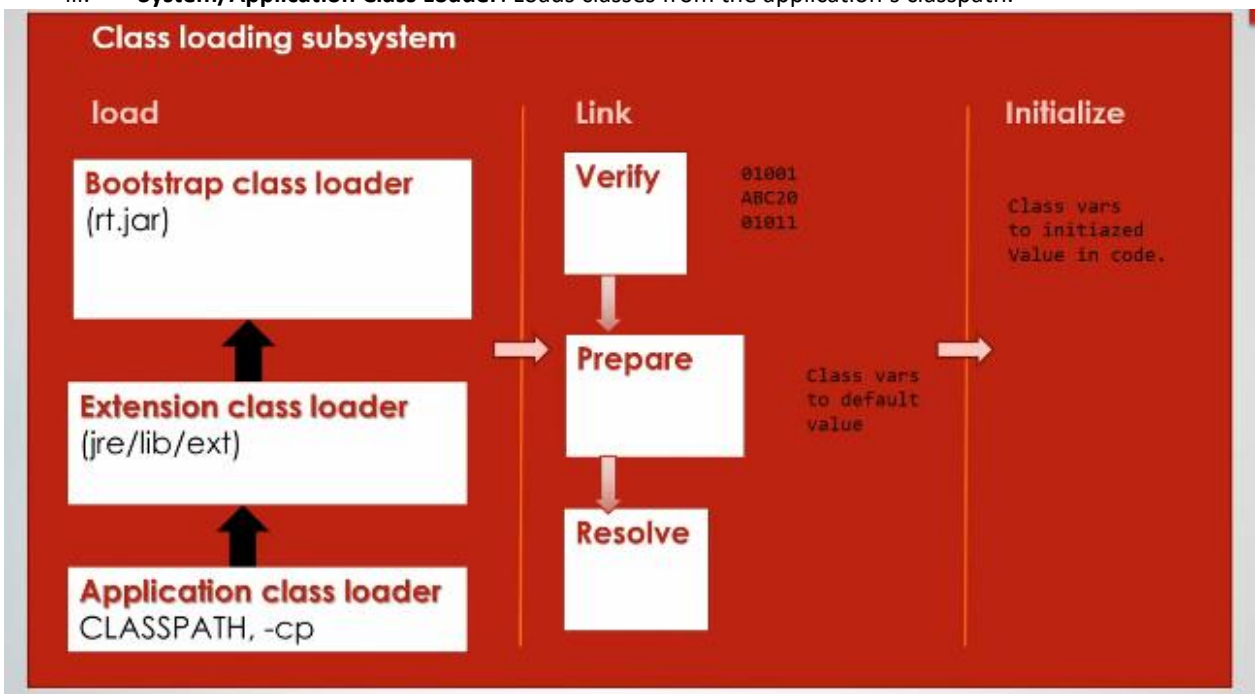
ii. **Linking**:
After loading, the JVM performs the linking process, which involves the following steps.
   - **Verification**: Ensures the correctness of the loaded class.
   - **Preparation**: JVM allocates memory for class variables and initializes them to default values.
   - **Resolution**: Transforms symbolic references from the type into direct references.

iii. **Initialization**:
This is the final phase of class loading, where the JVM assigns the correct initial values to the static fields of the class and executes any static initialization blocks.

4.1 Types of Class Loaders:

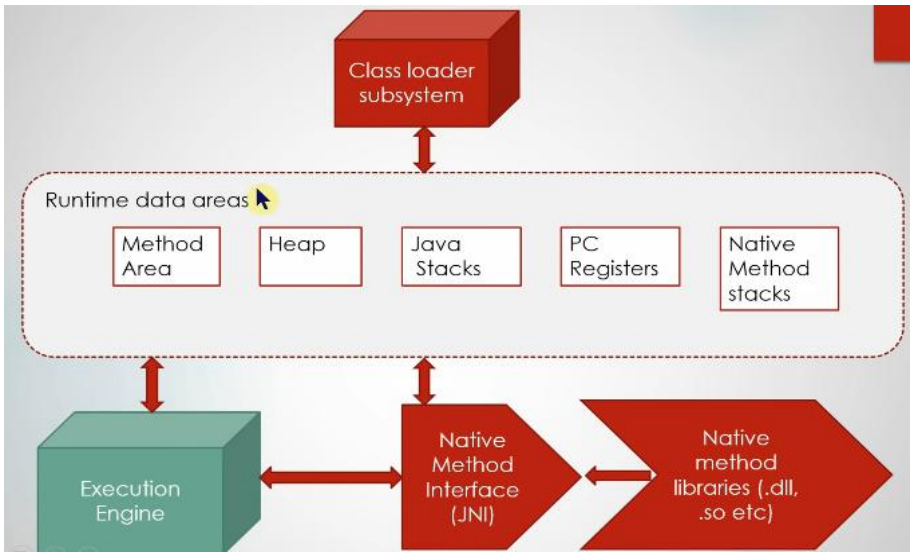
The JVM uses several types of class loaders in its operation:

i. **Bootstrap Class Loader**: Loads the core Java classes (like java.lang.Object).
ii. **Extension Class Loader**: Loads classes from the extensions directory of the Java Runtime Environment (JRE).
iii. **System/Application Class Loader**: Loads classes from the application's classpath.



5. JVM Architecture:

A Java 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.



5.1 Runtime Data Area:

The Runtime Data Area is divided into five major components:

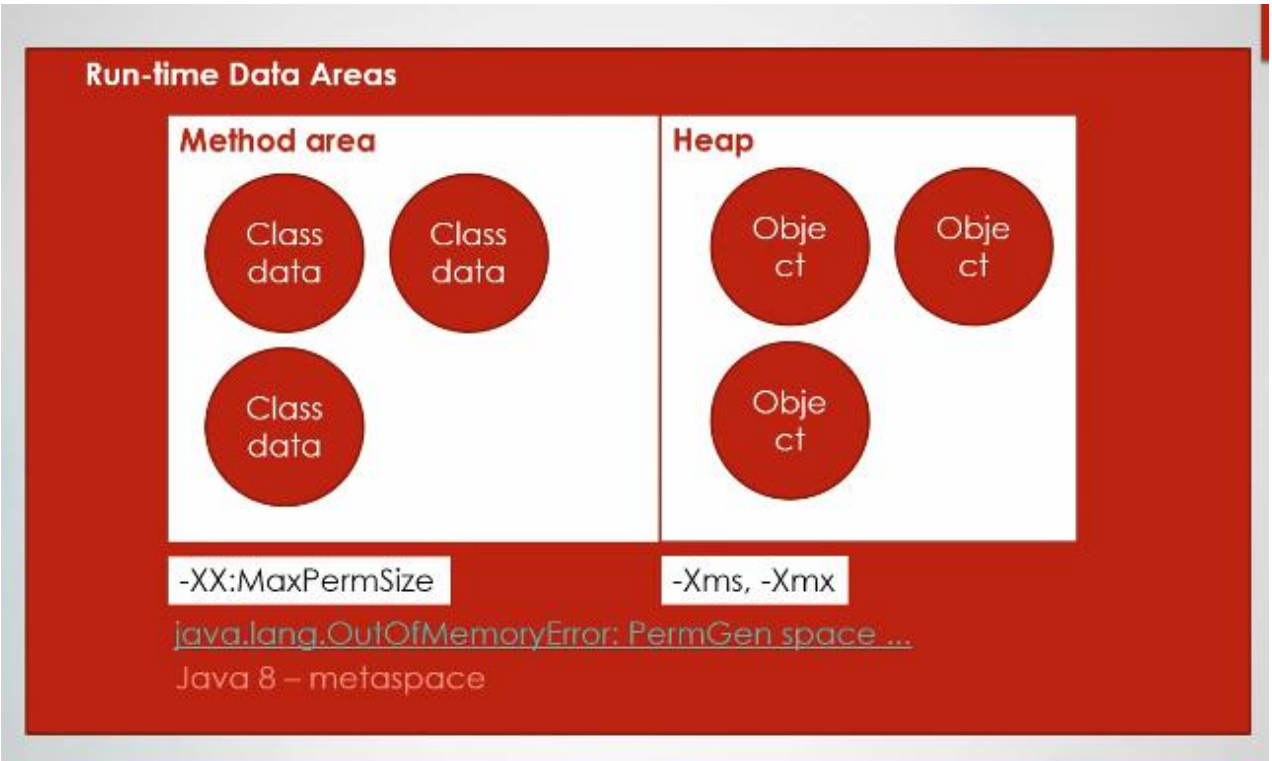
i. **Method area**:
In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

ii. **Heap area**:
Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.

iii. **Stack area**:
For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

iv. **PC Registers**:
Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.

v. **Native method stacks**:
For every thread, a separate native stack is created. It stores native method information.



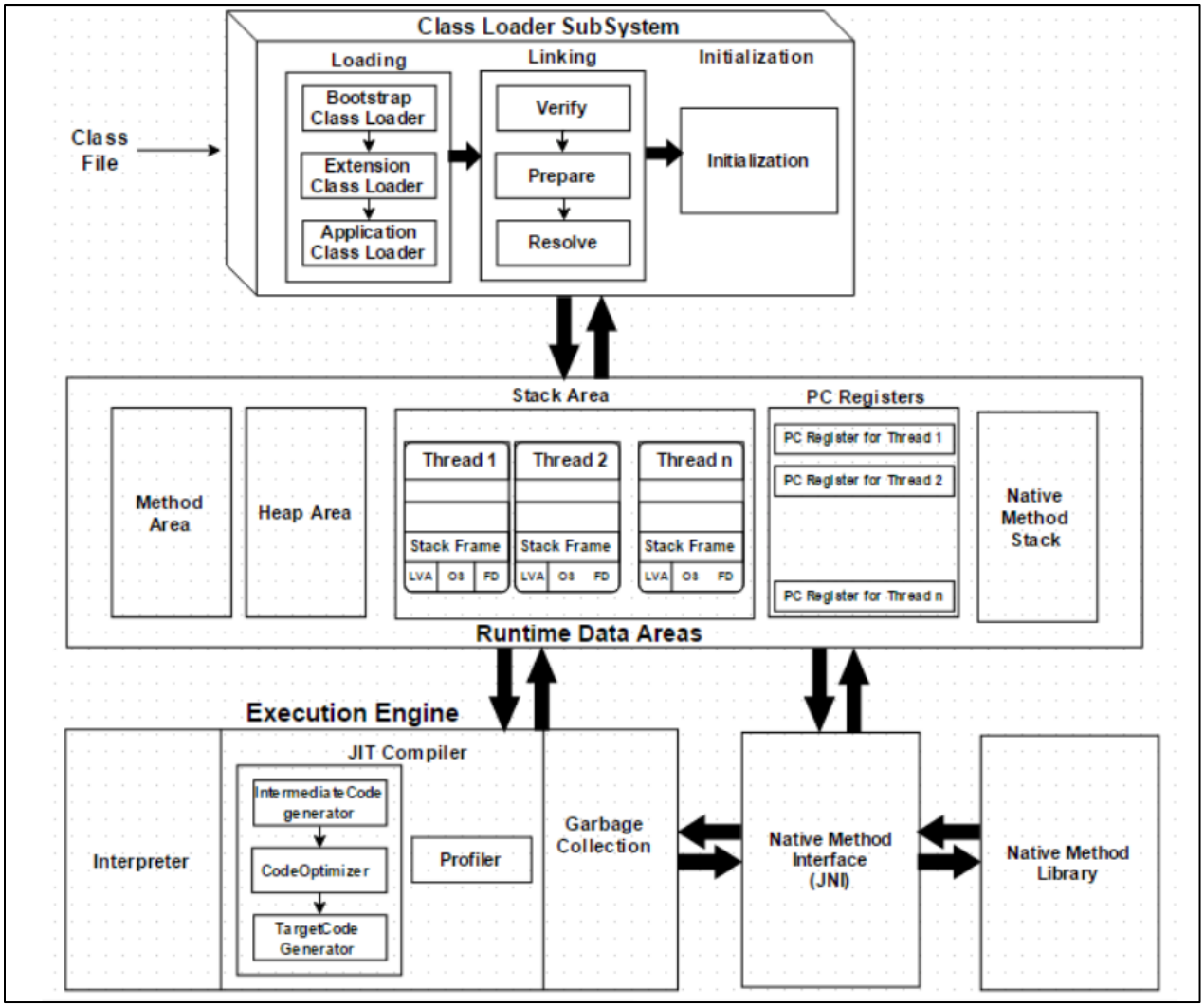
5.2 Execution Engine:

Execution engine executes the ".class" (bytecode). It reads the byte-code line by line, uses data and information present in various memory areas and executes instructions. It can be classified into **three** parts:

i. **Interpreter**:
It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

ii. **Just-In-Time Compiler (JIT)**:
It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part, so re-interpretation is not required, thus efficiency is improved. JIT compilers store its output in the **code cache** area for reuse.

iii. **Garbage Collector**:
It destroys unreferenced objects. For more on Garbage Collector, refer Garbage Collector.



Good to know:

JIT Compiler Optimization Techniques: The JIT compiler uses various optimization techniques to enhance performance:

- **Method In-lining:** To reduce the overhead of method calls, the JIT compiler often in-lines methods.
- **Dead Code Elimination:** Removes code segments that will never be executed.
- **Loop Optimization:** Enhances the performance of loops in several ways like unrolling.

6. Java Memory Management:

Java memory management is an automatic process that is managed by the Java Virtual Machine (JVM) itself. Java, being a block-structured language, uses a model where its memory is divided into two main types: stack and heap. JVM memory is divided into multiple parts:

i. **Non-Heap Memory:**

The Java Virtual Machine has memory other than the heap, referred to as Non-Heap Memory, also called off-heap. Off-heap memory refers to memory that is allocated outside of the JVM heap. It has following advantages.

- GC runs on heap memory. No GC for Off heap space. By using off heap space, we can reduce GC overhead.
- Efficient is handling Large Data Sets as no big pause for GC.
- JVM has an upper limit for heap memory. Off heap can be used to bypass these limits and we can use the total available system memory.

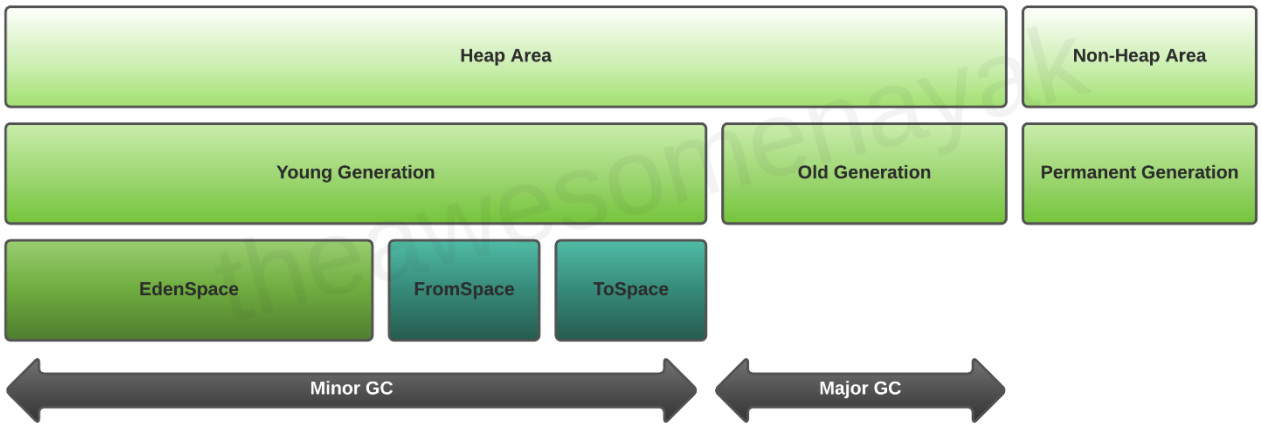
How programmatically can you use non-heap memory.

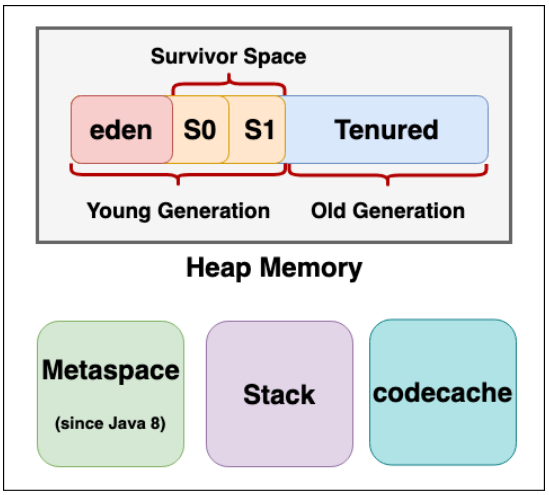
```
//this is example of java.nio
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

//Good to know-
//How to use off heap space in spark for caching?
SparkConf conf = new SparkConf().
set("spark.memory.offHeap.enabled","true").set("spark.memory.offHeap.size","2g");
Dataset<Row> df = spark.read().json("<file path>")
df.persist(StorageLevel.OFF_HEAP);
```

Java Virtual Machine uses this space to store the JVM code itself, JVM internal structures, loaded profiler agent code, and data, etc.

ii. **Heap Memory:**

Heap memory is part of runtime data area and JVM is responsible for dynamically allocating and managing objects during program execution in this area. The heap is created when the Java Virtual Machine starts up and may increase or decrease in size while the application runs. The size of the heap can be specified using –Xms VM option. The heap can be of fixed size or variable size depending on the garbage collection strategy. Maximum heap size can be set using –Xmx option.





iii. **Young Generation**

Newly created objects start in the Young Generation. The Young Generation is further subdivided into:

- **Eden space -** all new objects start here, and the initial memory is allocated to them
- **Survivor spaces (FromSpace, ToSpace) -** objects are moved here from Eden after surviving one garbage collection cycle.

When objects are garbage collected from the Young Generation, it is a minor garbage collection event.
When Eden space is filled with objects, a Minor GC is performed. All the dead objects are deleted, and all the live objects are moved to one of the survivor spaces. Minor GC also checks the objects in a survivor space and moves them to the other survivor space.

iv. **Old Generation**

Objects that are long-lived (survived) are eventually moved from the Young Generation to the Old Generation. This is also known as Tenured Generation and contains objects that have remained in the survivor spaces for a long time.

There is a threshold defined for the tenure of an object which decides how many garbage collections cycles it can survive before it is moved to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.

v. **Permanent Generation**

Metadata such as classes and methods are stored in the Permanent Generation. It is populated by the JVM at runtime based on classes in use by the application. Even though PermGen is not part of the main heap, it is included in a full garbage collection. Classes may get unloaded (garbage collected) if they are no longer needed and space is required for other classes. Classes that are no longer in use may be garbage collected from the Permanent Generation.

You can use the -XX:PermGen and -XX:MaxPermGen flags to set the initial and maximum size of the Permanent Generation.

vi. **MetaSpace**

Starting with Java 8, the MetaSpace memory space replaces the PermGen space. The implementation differs from the PermGen, and this space of the heap is now automatically resized. This avoids the problem of applications running out of memory due to the limited size of the PermGen space of the heap. The MetaSpace memory can be garbage collected and the classes that are no longer used can be automatically cleaned when the MetaSpace reaches its maximum size.

vii. **Code Cache:**

JVM has an interpreter to interpret the byte code and convert it into hardware dependent machine code. As part of JVM optimization, the Just in Time (JIT) compiler has been introduced. The frequently accessed code blocks will be compiled to native code by JIT and stored in code cache. The JIT compiled code will not be interpreted.

viii. **Meta Space:**

This memory is out of heap memory and part of the native memory. As per the document by default the meta space doesn't have an upper limit. In earlier versions of Java, we called this "***Perm Gen Space***". This space is used to store class definitions loaded by class loaders. This is designed to grow in order to avoid out-of-memory errors. However, if it grows more than the available physical memory, then the operating system will use virtual memory. This will have an adverse effect on application performance, as swapping the data from virtual memory to physical memory and vice versa is a costly operation. We have JVM options to limit the Meta Space used by JVM. In that case, we may get out of memory errors.

Good to know

General cycle - When an object is created, it is first put into the Eden space of the young generation. Once a minor garbage collection happens, the live objects from Eden are promoted to the FromSpace(S1). When the next minor garbage collection happens, the live objects from both Eden and FromSpace are moved to the ToSpace(S2). This cycle continues for a specific number of times. If the object is still used after this point, the next garbage collection cycle will move it to the old generation space.

6.1 Major vs Minor GC

Minor GC: A garbage collection event that happens in the Young Generation of the heap (Eden + Survivor spaces).

Has following Activities—

- Removes short-lived objects.
- Moves surviving objects to Survivor space or Old Generation.
- Short pauses (milliseconds), usually **stop-the-world (STW)** but fast.

Major GC: A garbage collection event that affects the Old Generation (and sometimes the entire heap).

Has following Activities—

- Cleans up long-lived objects.
- Often involves **compacting** the heap.
- Much longer than Minor GC — can be **hundreds of milliseconds to seconds** depending on heap size and GC algorithm.

Note: Frequent Major GCs often indicate memory leaks or insufficient heap size.
Monitoring GC logs (-Xlog:gc in Java 17) helps identify and tune GC behavior.*

7. Garbage Collections:

Garbage Collection is the process of reclaiming the runtime unused memory by destroying the unused objects. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Over the lifetime of a Java application, new objects are created and released. Eventually, some objects are no longer needed. You can say that at any point in time, the heap memory consists of two types of objects:

- **Live** - these objects are being used and referenced from somewhere else

- **Dead** - these objects are no longer used or referenced from anywhere

The garbage collector finds these unused objects and deletes them to free up memory.
The garbage collection implementation lives in the JVM. Each JVM can implement its own version of garbage collection.

There are various ways in which references to an object can be released to make it a candidate for Garbage Collection. For example, making a reference null, passing anonymous object to method, assigning a reference to another.

> **Good to know**
> JVM is a specification, any vendor(company) can implement JVM by their own implementation along with own garbage collection implementation.

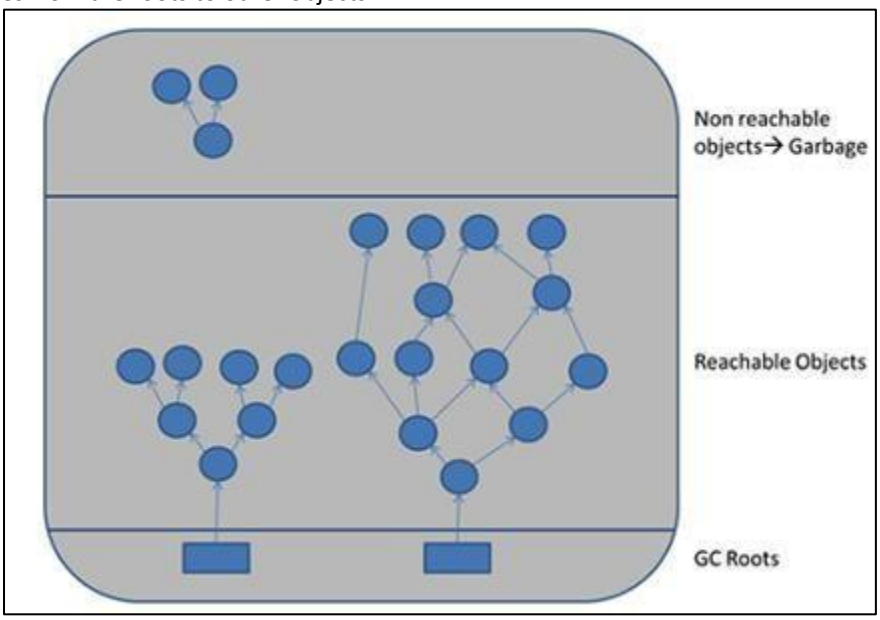
7.1 Garbage Collection Roots

GC Roots are the source of all Object Trees. They are special objects for the garbage collector. GC roots are entry points of the graph of Objects in memory for the garbage collector processes. Objects that are reachable through a chain of references from GC roots are considered live objects, and those that are not considered candidates for garbage collection.

7.2 Types of GC Roots

- **Class**: Classes loaded by a system class loader. contains references to static variables as well.
- **Stack Local**: Local variables and parameters to methods stored on the local stack.
- **Active Java Threads**: All active Java threads
- **JNI References**: Native code Java objects created for JNI calls, contain local variables, parameters to JNI methods, and global JNI references.

The garbage collector traverses the whole object graph in memory, starting from those Garbage Collection Roots and following references from the roots to other objects.

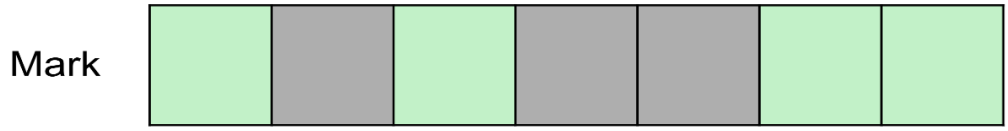


7.3 Phases of Garbage Collection in Java

A standard Garbage Collection implementation involves three phases:

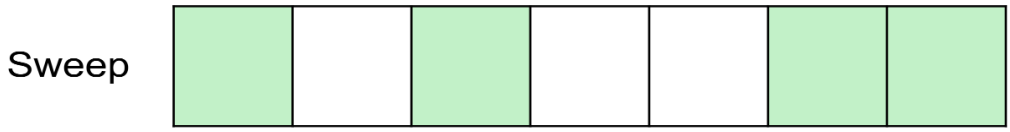
i. **Mark objects:**

GC marks all the live objects in memory by traversing the object graph. All the objects which are not reachable from GC Roots are garbage and considered as candidates for garbage collection.



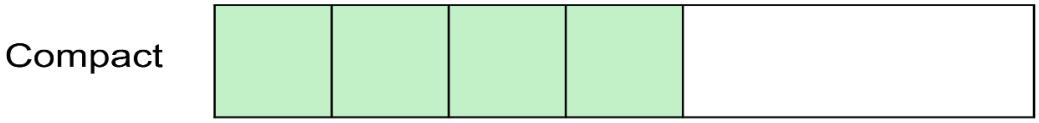
ii. **Sweep objects:**

After marking phase, we have the memory space which is occupied by live (visited) and dead (unvisited) objects. The sweep phase releases the memory fragments which contain these dead objects.



iii. **Compact**:

The dead objects that were removed during the sweep phase may not necessarily be next to each other. Thus, you can end up having fragmented memory space. Memory can be compacted after the garbage collector deletes the dead objects, so that the remaining objects are in a contiguous block at the start of the heap.

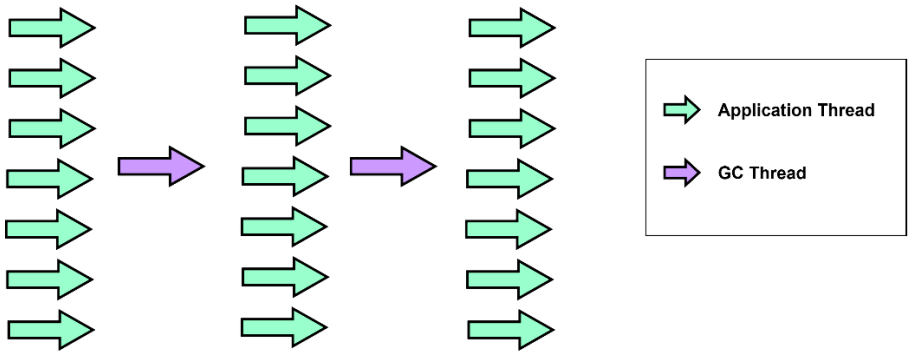


7.4 Types of Garbage Collectors in the Java Virtual Machine

Garbage collection makes Java memory efficient because it removes the unreferenced objects from heap memory and makes free space for new objects. The Java Virtual Machine has the following types of garbage collectors. Let's look at each one in detail.

i. Serial GC

This is the simplest implementation of GC and is designed for small applications running on single-threaded environments. All garbage collection events are conducted serially in one thread. Compaction is executed after each garbage collection.



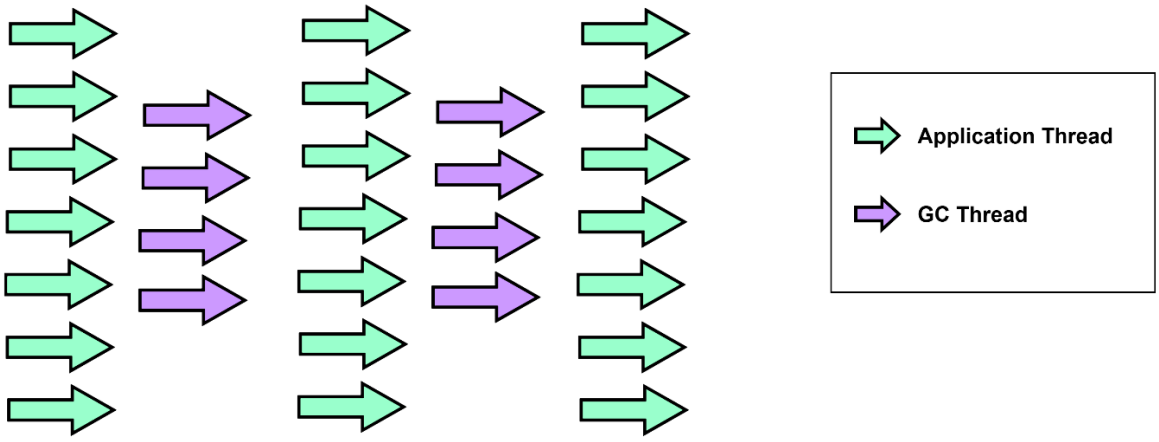
When it runs, it leads to a "stop the world" event where the entire application is paused. Since the entire application is frozen during garbage collection, it is not recommended in a real-world scenario where low latencies are required.

The JVM argument to use the Serial Garbage Collector is -XX: +UseSerialGC.

ii. Parallel GC

The parallel collector is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. This is the default implementation of GC in the JVM and is also known as Throughput Collector.

Multiple threads are used for minor garbage collection in the Young Generation. A single thread is used for major garbage collection in the Old Generation.



Running the Parallel GC also causes a "stop the world event" and the application freezes. Since it is more suitable in a multi-threaded environment, it can be used when a lot of work needs to be done and long pauses are acceptable, for example running a batch job.

The JVM argument to use the Parallel Garbage Collector is -XX: +UseParallelGC.

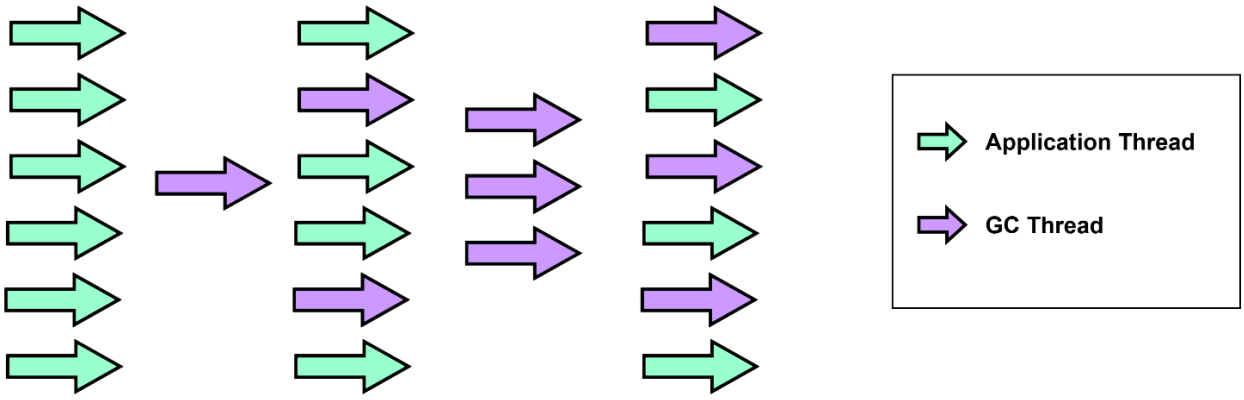
iii. Parallel Old GC

This is the default version of Parallel GC since Java 7u4. It is the same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation.

The JVM argument to use Parallel Garbage Collector is -XX: +UseParallelOldGC.

iv. CMS (Concurrent Mark Sweep) GC

This is also known as the concurrent low pause collector. Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old GC, but CMS runs concurrently alongside application processes to minimize “stop the world” events.



Because of this, the CMS collector uses more CPU than other GCs. If you can allocate more CPU for better performance, then the CMS garbage collector is a better choice than the parallel collector. No compaction is performed in CMS GC.

The JVM argument to use Concurrent Mark Sweep Garbage Collector is -XX: +UseConcMarkSweepGC.

v. G1 (Garbage First) GC

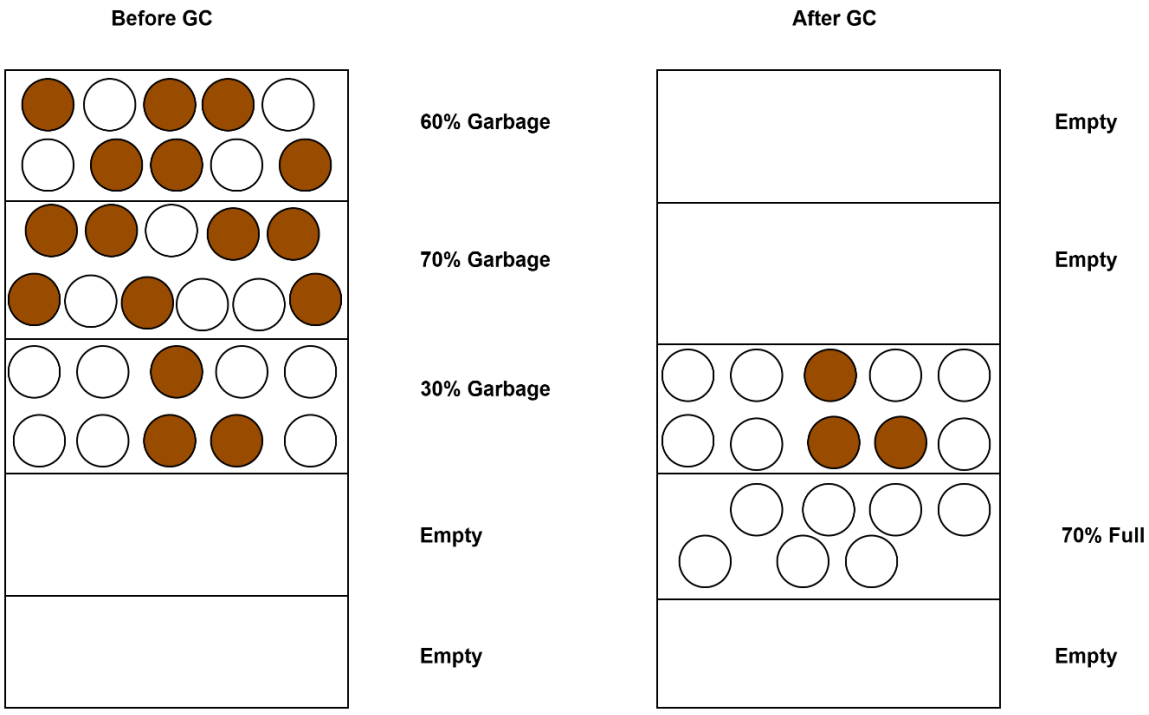
G1GC was intended as a replacement for CMS and was designed for multi-threaded applications that have a large heap size available (more than 4GB). It is parallel and concurrent like CMS, but it works quite differently under the hood compared to the older garbage collectors.

Although G1 is also generational, it does not have separate regions for young and old generations. Instead, each generation is a set of regions, which allows resizing of the young generation in a flexible way.

It partitions the heap into a set of equal size regions (1MB to 32MB – depending on the size of the heap) and uses multiple threads to scan them. A region might be either an old region or a young region at any time during the program run.

After the mark phase is completed, G1 knows which regions contain the most garbage objects. If the user is interested in minimal pause times, G1 can choose to evacuate only a few regions. If the user is not worried about pause times or has stated a fairly large pause-time goal, G1 might choose to include more regions.

Since G1GC identifies the regions with the most garbage and performs garbage collection on that region first, it is called Garbage First.



Apart from the Eden, Survivor, and old memory regions, there are two more types of regions present in the G1GC:

- *Humongous* - used for large size objects (larger than 50% of heap size)
  - Such an object is allocated directly in the old generation into "Humongous regions". These Humongous regions are a contiguous set of regions.
- *Available* - unused or non-allocated space

The JVM argument to use the G1 Garbage Collector is -XX: +UseG1GC.

8. Java Collections –

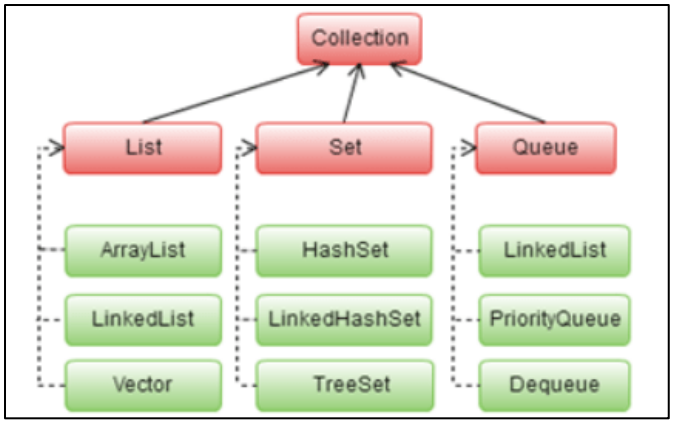
A collection in Java is a group of individual objects that are treated as a single unit. The Java Collections Framework (JCF) is a unified architecture for storing, retrieving, and manipulating groups of objects in Java.

8.1 Collection Interface

This is the root interface for all collection types (except maps). It has following methods:
add(), remove(), size(), iterator().

The Collection interface in Java is having following sub Interfaces:

- **List**: Ordered, allows duplicates (ArrayList, LinkedList).
- **Set**: Unordered, no duplicates (HashSet, LinkedHashSet, TreeSet).
- **Queue**: FIFO ordering (PriorityQueue, LinkedList).



8.2 Collection Implementations—

8.2.1 List Implementations

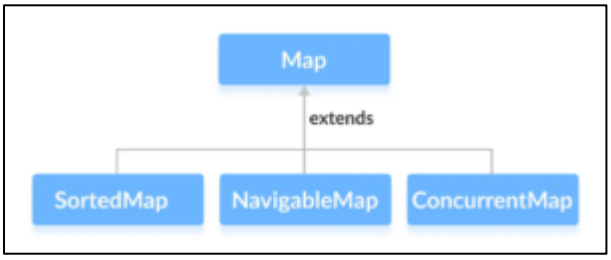
- ArrayList
  - Backed by a dynamic array.
  - Fast random access, slow insertion/removal in the middle.
- LinkedList
  - Doubly linked nodes.
  - Fast insert/delete in middle, slower random access.

8.2.2 Set Implementations

- **HashSet**: Uses hash table, no order guarantee.
- **LinkedHashSet**: Maintains insertion order.
- **TreeSet**: Sorted, based on TreeMap (Red-Black Tree).

8.2.3 Queue & Deque Implementations

- **PriorityQueue**: Elements ordered by priority.
- **ArrayDeque**: Fast double-ended queue.



8.3 Map Implementations

- **HashMap**: Hash table, O(1) average time complexity.
- **LinkedHashMap**: Maintains insertion order.
- **TreeMap**: Sorted keys, log(n) access time.
- **ConcurrentHashMap**: Thread-safe, high concurrency.

8.4 Collections class:

Collections is a utility class present in java.util package. It defines several utility methods like sorting and searching which is used to operate on collection. It has all static methods.

8.4.1 Collection vs Collections –

Feature	Collection	Collections
Type	Interface	Utility Class
Package	java.util	java.util
Purpose	Defines a group of objects (data holder)	Provides utility methods for collections
Contains	Abstract methods	Static methods
Example	List, Set, Queue	Collections.sort(list), Collections.max(list)
Extensible	Yes (implement it)	No (final class)

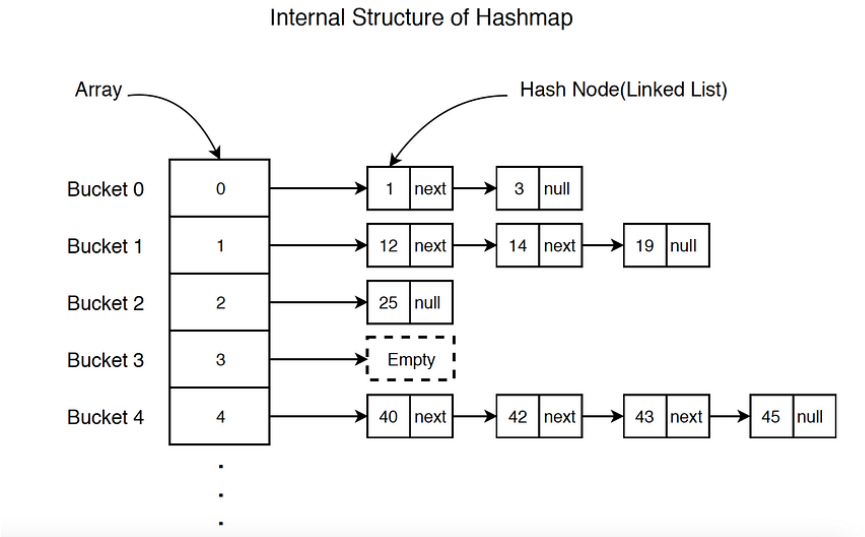
8.4.2 Use cases—

- ArrayList → best for frequent reads, fewer inserts/removes
- LinkedList → best for frequent inserts/removes, fewer reads
- HashMap/HashSet → constant-time operations if hash function is good
- TreeMap/TreeSet → sorted order but O(log n) performance

8.4.3 Hashmap internal:

- HashMap is part of Java Collections framework and is used to store key-value pairs. It is part of java.util package. Very useful unsynchronized data structure when you want to Insert / Delete / Lookup values in constant time O(1).
- Array of Nodes (Buckets)
- Each bucket holds a linked list or a balanced tree (red-black tree) if too many collisions happen.

```
/*
 * This is generic class of HashMap
 * Where we can insert any type of object as key & value.
 * next is reference pointer to node as we know hashmap creates
 * linkedlist when multiple key-value pairs collide on same bucket index.
*/
private Entry<K,V>[] buckets;
//I've init it to default capacity of bucket i.e. 16
private int capacity = 16;

class Entry<K,V> {
 K key;
 V value;
 Entry<K, V> next;

 public Entry( K key, V value ) {
   this.key = key;
   this.value = value;
   this.next = null;
 }
}
```



8.4.4 Internal working—

- Calculate hash code of key.
- If basket with that hash code is present then use **equals** method on the key, search the keys in the basket to determine if element is be added or replaced.
- If not present, then add new basket (rehashing) and add entry to it.

9. Reference Classes in java –

Memory management is done automatically in Java. The programmer doesn't need to worry about reference objects that have been released. One downside to this approach is that the programmer cannot know when a particular object will be collected. Moreover, the programmer has no control over memory management. However, the **java.lang.ref package** defines classes that provide a limited degree of interaction with the garbage collector. The concrete classes SoftReference, WeakReference and PhantomReference are subclasses of Reference that interact with the garbage collector in different ways.

This is the superclass for reference classes like SoftReference, WeakReference, and PhantomReference.

Though both WeakReference and SoftReference helps garbage collector and memory efficient, WeakReference becomes eligible for garbage collection as soon as last strong reference is lost but SoftReference even though it cannot prevent GC, it can delay it until JVM absolutely need memory.

9.1 Reference Types –

We have different types of references: **strong, weak, soft, and phantom references**.

The difference between the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria.

9.1.1 Strong Reference:

StringBuilder builder = new StringBuilder();

9.1.2 WeakReference:

WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());

A nice use case for weak references is caching scenarios. Imagine that you retrieve some data, and you want it to be stored in memory as well — the same data could be requested again. On the other hand, you are not sure when, or if,

this data will be requested again. A nice implementation for caching scenarios is the collection WeakHashMap<K,V>. If we open the WeakHashMap class in the Java API, we see that its entries actually extend the WeakReference class and uses its ref field as the map's key:

```
private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {
   V value;
```

Once a key from the WeakHashMap is garbage collected, the entire entry is removed from the map.

9.1.3 Soft Reference

These types of references are used for more memory-sensitive scenarios

since those are going to be garbage collected only when your application is running low on memory.

Java guarantees that all soft referenced objects are cleaned up before it throws an OutOfMemoryError.

```
SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());
```

9.1.4 Phantom Reference

Phantom Reference can be used in situations, where sometime using *finalize()* is not sensible thing to do. This reference type differs from the other types defined in java.lang.ref Package because it isn't meant to be used to access the object, but as a signal that the object has already been finalized, and the garbage collector is ready to reclaim its memory.

These types of references are considered preferable to finalizers.

We can't get a referent of a phantom reference. The referent is never accessible directly through the API and this is why we need a reference queue to work with this type of references.

The Garbage Collector adds a phantom reference to a reference queue **after the finalize method of its referent is executed**. It implies that the instance is still in the memory.

People usually attempt to use finalize() method to perform postmortem cleanup on objects which usually not advisable. As mentioned earlier, Finalizers have an impact on the performance of the garbage collector since Objects with finalizers are slow to garbage collect.

Phantom references are safe way to know an object has been removed from memory. For instance, consider an application that deals with large images. Suppose that we want to load a big image in to memory when large image is already in memory which is ready for garbage collected. In such case, we want to wait until the old image is collected before loading a new one. Here, the phantom reference is flexible and safely option to choose. The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory. Similarly, we can use Phantom References to implement a Connection Pool. We can easily gain control over the number of open connections, and can block until one becomes available.

10. Multitasking:

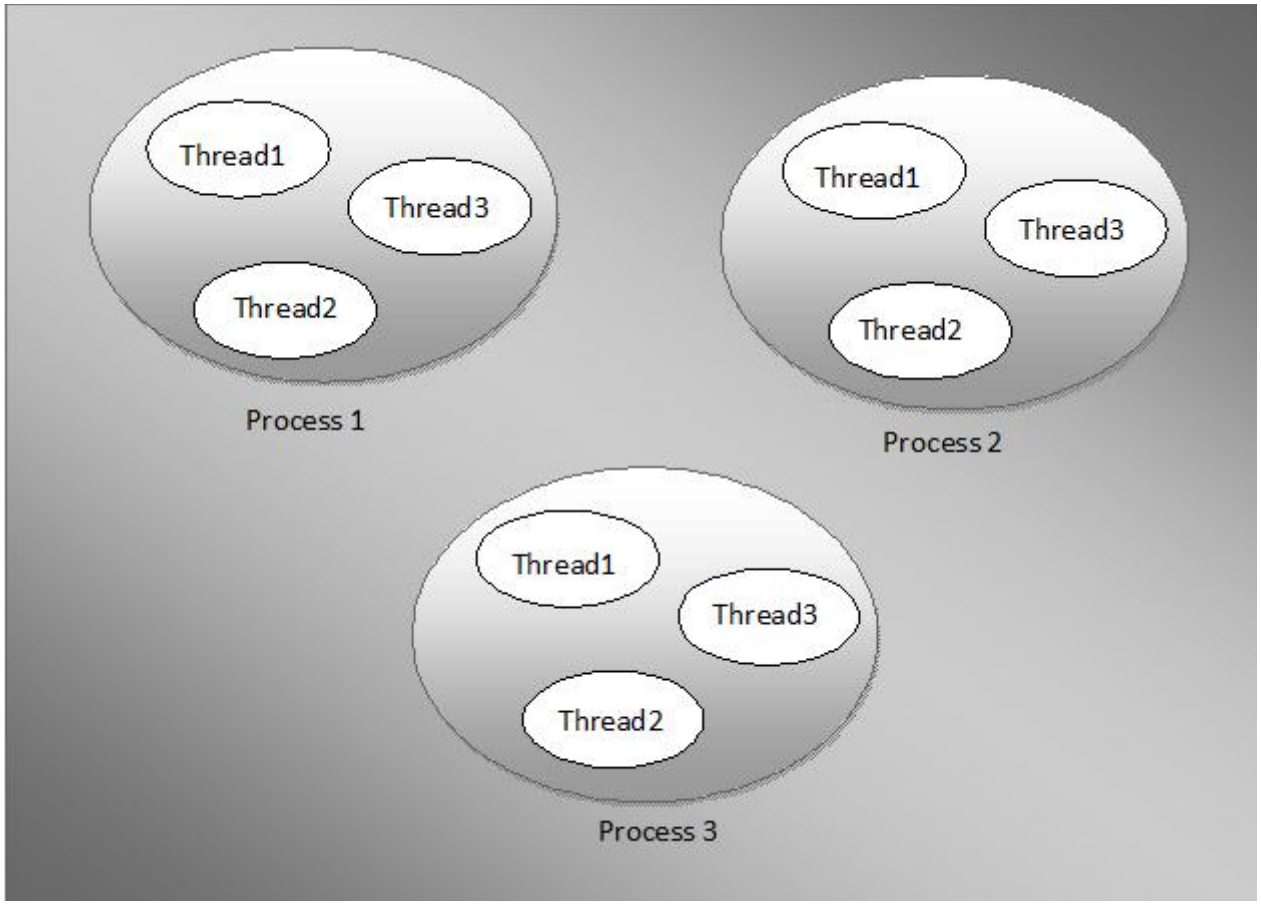
Multitasking is the ability to do more than one thing at the same time. This is called Multitasking and software that can do such things is known as **concurrent software.**

10.1 Type of Multitasking:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

In process-based multitasking, two or more processes or programs can be run concurrently. In thread-based multitasking, two or more threads can be run concurrently. Each process has an address in memory. In other words, each process allocates a separate memory area. Threads share the same address space.

Process based multitasking example: - Running music on background while working on text editor.
Thread based multitasking example: - Typing on word document and printing out simultaneously.
Executing multiple threads simultaneously is known as **multithreading**.



In the above figure, Threads exist within a process and every process has at least one thread.

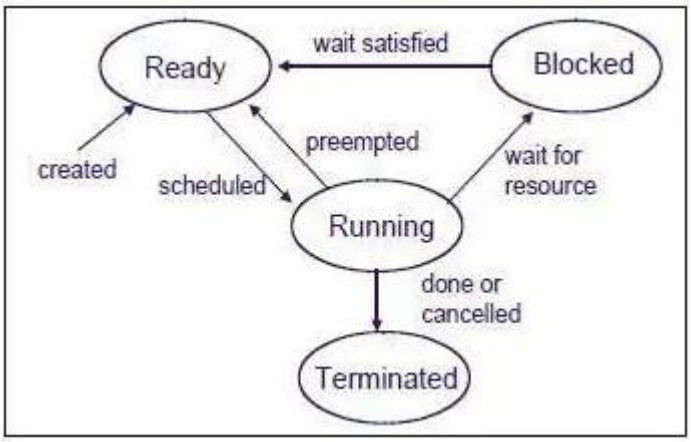
Process	Thread
Processes are heavy weight operations.	Threads are light weight operations.
Every process has its own memory space.	Threads use the memory of the process they belong to.
Inter process communication is slow as processes have different memory address.	Inter thread communication is fast as threads of the same process share the same memory address of the process they belong to.
Context switching between the process is more expensive.	Context switching between threads of the same process is less expensive.
Processes don't share the memory with other processes.	Threads share the memory with other threads of the same process.

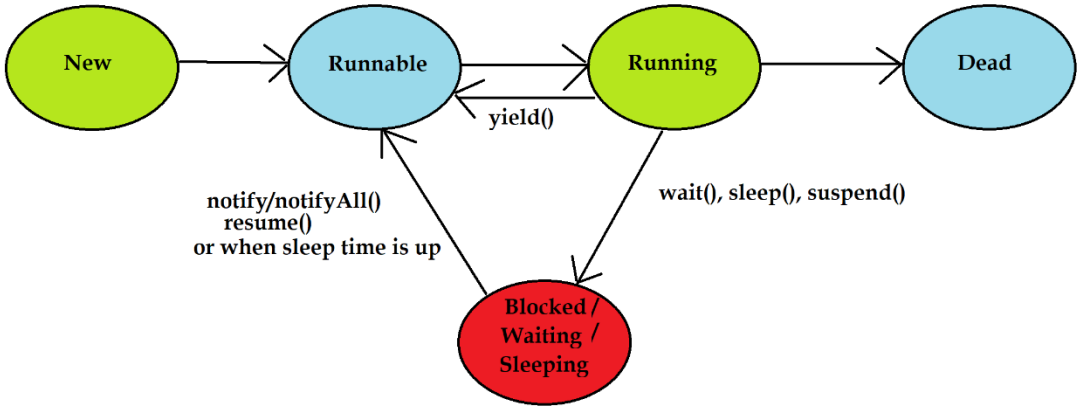
10.2 Threads in Java

In a Java program, every program has one or more threads. There are many threads that are there by default, such as garbage collector thread to perform GC, main thread etc. We can say threads are simply tasks that we want to execute parallelly.

10.3 Lifecycle of a Thread

- creation of a Thread.
- Starting a Thread.
- running a Thread.
- pausing, suspending, and resuming the Threads.
- terminating Thread.





Thread Lifecycle using Thread states

There are two basic strategies for using Thread objects to create a concurrent application.

(i) Using Thread Class

```
HelloThread t = new HelloThread();
```

(ii) Using Runnable Interface

```
package ThreadLearning;

public class HelloRunnable
                implements Runnable
{
    public void run()
    {
        System.out.println("Helloo you are inside run method");
    }


    public static void main(String[] args)
    {
        System.out.println("Inside the main method");

        (new Thread(new HelloRunnable())).start();

        Thread thread1 = new Thread(new HelloRunnable());
        thread1.start();

        Thread thread2 = new Thread(new HelloRunnable(), "Thread2");
        thread2.start();
    }
}
```

10.4 Thread Scheduler:

Scheduler is the part of JVM that decides which thread should be picked first. In a single processor machine, only one thread can run at a time. And hence it is the thread scheduler that decides which thread (among all the eligible threads) will actually run. When we say eligible, it means in the runnable state (ready-to-run). Any thread in the runnable state can be chosen by the scheduler to be the one and only running thread. Thread scheduler can move a thread from the running state back to runnable state also.

The order in which runnable threads are chosen is not guaranteed. Schedulers in JVM implementations usually employ one of the following two strategies.

- Preemptive scheduling.
- Time-Sliced or Round-Robin scheduling.

10.5 Thread Methods

10.5.1 **start**:

Thread can be started by calling its start () method. This method performs some internal housekeeping and calls the thread's run () method. When start() method returns, two threads are now executing in parallel: the original thread (which has returned from calling the start () method) and the newly started thread (which is now executing its run() method).

10.5.2 **sleep**:

Thread.sleep() causes the current thread to suspend execution for a specified period (no lock released if acquired).

10.5.3 **Join():**

The join method allows one thread to wait for the completion of another (joining thread). f t is a thread which is currently executing, then t.join() will join in the execution path of calling thread(by default main thread).

```
t.join();
```

```
public static void main(String args[])
    throws InterruptedException
{
    //(new HelloThread()).start();
    System.out.println("Starting of main method");
    HelloThread t = new HelloThread();
    t.start();
    if(t.isAlive())
    {
        System.out.println("**********Thread is still alive**********");
    }
    t.join();
    System.out.println("Exiting from main method");
```

In the above example "Exiting from the main method" will not be printed until thread t complete its execution. Hence 't' thread forced the current thread (main thread) to pause its execution until its execution is terminated or completed.

Like sleep, join responds to an interrupt by exiting with an InterruptedException. A call to join() method **guaranteed** to cause the current thread to stop executing until the thread it joins with completes.

- **Threads Communication**

Threads always run with some priority, which is usually represented by a number between 1 and 10. If thread enters the Runnable state, and it has a higher priority than any of the threads in the pool and higher than the currently running thread, then lower priority running thread will be moved back to Runnable state and highest priority thread will be chosen to run. In nutshell running thread will be of equal or greater priority than the highest priority threads in the pool. We can set the priority of the thread by directly calling setPriority() method. The default priority of the thread is 5.

10.5.4 yield():

this method will **try** to move the currently running thread back to Runnable state to allow other threads of same priority to run. If any thread executes the yield() method, the thread scheduler checks if there is any thread with the same or high priority as this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing. That is why it's **not guaranteed** operation.

10.5.5 wait():

wait method is a part of **java.lang.Object** class. When wait() method is called, the calling thread stops its execution until notify() or notifyAll() method is invoked by some other Thread.

10.5.6 notify():

The notify() method is defined in the **java.lang.Object** class. It is used to wake up only one thread that's waiting for an object, and that thread then begins execution.

10.6 Concurrency:

Concurrency refers to the ability of a program to execute multiple tasks simultaneously. It enables the efficient utilization of system resources and can improve the overall performance and responsiveness of the application.
Multithreading is just one way to achieve concurrency in Java. Concurrency can also be achieved through other means, such as multiprocessing, asynchronous programming, or event-driven programming. is a great solution for building high-performance modern applications for a variety of reasons. Concurrency allows for the division of complex and time-consuming tasks into smaller parts that can be executed simultaneously, resulting in improved performance. This makes the most of today's multi-core CPUs and can make applications run much faster.

10.7 Concurrency Issues:

Threads co-ordinate with each other by sharing data (fields and object reference) which is extremely efficient form of communication between threads. But, since thread execution is asynchronous, the details of how threads interact can be unpredictable and can make two kinds of errors possible below

- Thread interference
- Memory consistency errors.

10.8 Thread interference:

Thread Interference describes how errors are introduced when multiple threads access shared data.
Consider a simple class called Counter as shown below.

```
package ThreadLearning;

public class Counter
{
    private int c = 0;

    public void increment()
    {
        c++;
    }

    public void decrement()
    {
        c--;
    }

    public int value()
    {
        return c;
    }
}
```

Counter is designed such that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Synchronization is a tool which helps to prevent these errors. The Java programming language provides two basic synchronization idioms:

- Synchronized methods
- Synchronized statements.

10.9 Executor:

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.
This is an interface and provides a way to execute submitted Runnable tasks.

10.10 Executors:

Java executor framework (java.util.concurrent.Executor), is used to run the Runnable objects without creating new threads every time and mostly re-using the already created threads.

The java.util.concurrent.Executors provide factory methods that are being used to create ThreadPools of worker threads.

Some types of Java Executors are listed below:

- SingleThreadExecutor
- FixedThreadPool(n)
- CachedThreadPool
- ScheduledExecutor

SingleThreadExecutor: Create a thread pool having single thread. All task executed sequentially.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

FixedThreadPool: It is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads and if there is more task, they are stored on a LinkedBlockingQueue. It uses Blocking Queue.

CachedThreadPool:

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. It uses a SynchronousQueue queue.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

ScheduledExecutor:

This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

```
ScheduledExecutorService scheduledExecService = Executors.newScheduledThreadPool(1);
```

Thread pools overcome this issue by keeping the threads alive and reusing the threads. Any excess tasks flowing in, that the threads in the pool can't handle are held in a Queue. Once any of the threads get free, they pick up the next task from this queue.

Threads require some resources to start, and they are stopped after the task is done. For applications with many tasks, you would want to queue up tasks instead of creating more threads. Wouldn't it be great if we could somehow **reuse** existing threads while also limiting the number of threads you can create?

The **ExecutorService** class allows us to create a certain number of threads and distribute tasks among the threads. Since you are creating a fixed number of threads, you have a lot of control over the performance of your application.

10.11 Parallel Computing

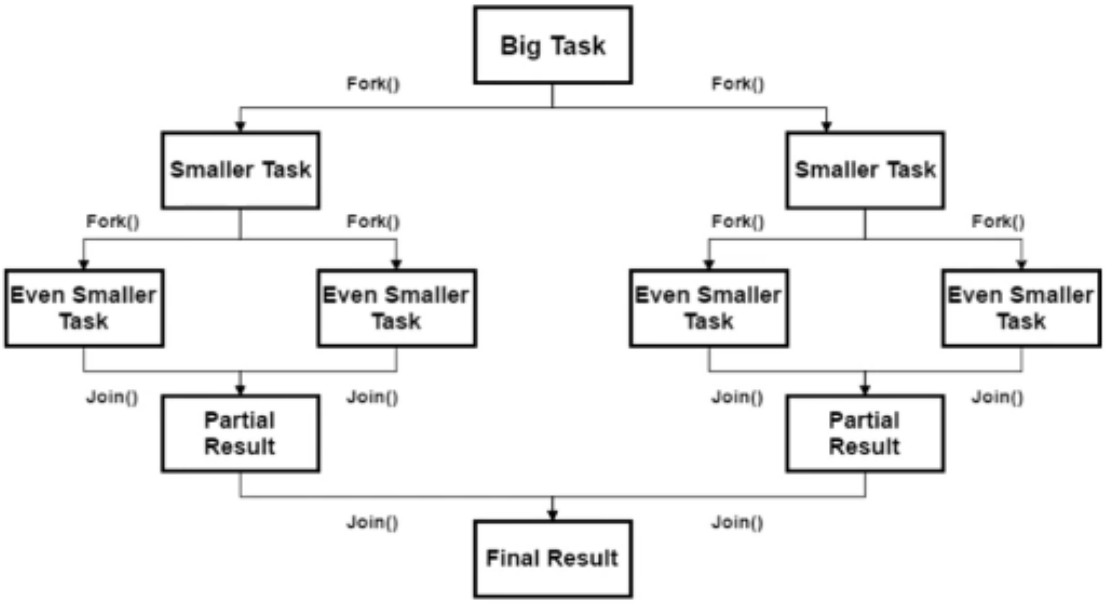
Parallel computing or parallelization is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). In essence, if a CPU intensive problem can be divided in smaller, independent tasks, then those tasks can be assigned to different processors.

10.12 Java Fork and Join using ForkJoinPool

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the Java ExecutorService but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task. It may sound a bit abstract, so in this fork and join tutorial I will explain how the ForkJoinPool works, and how splitting tasks up work.

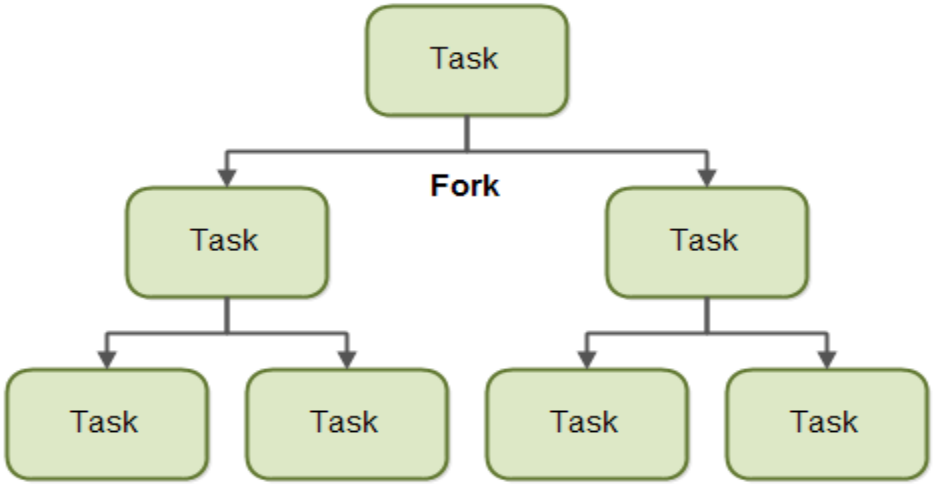
10.13 Fork and Join Explained

Before we look at the ForkJoinPool I want to explain how the fork and join principle works in general. The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.



10.14 Fork

A task that uses the fork and joins principle can fork (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:

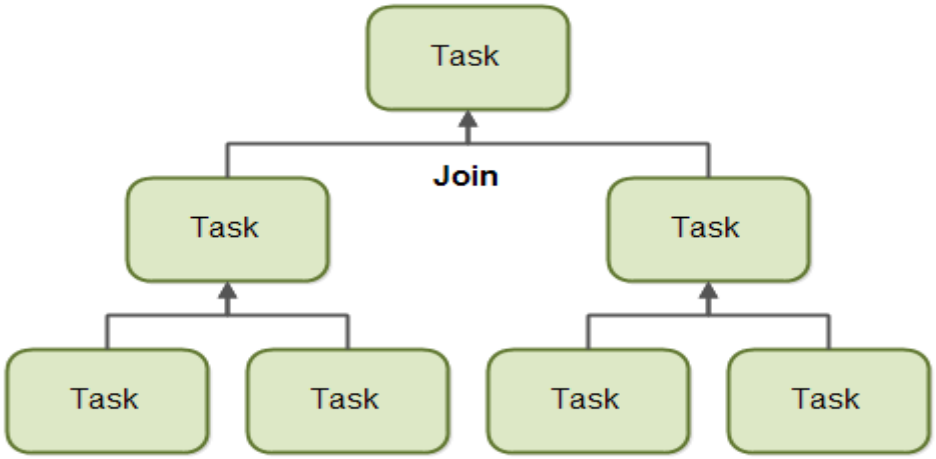


By splitting itself up into subtasks, each subtask can be executed in parallel by **different CPUs**, or **different threads on the same CPU**. A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.

The **limit** for when it makes sense to fork a task into subtasks is also called a **threshold**. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

10.15 Join

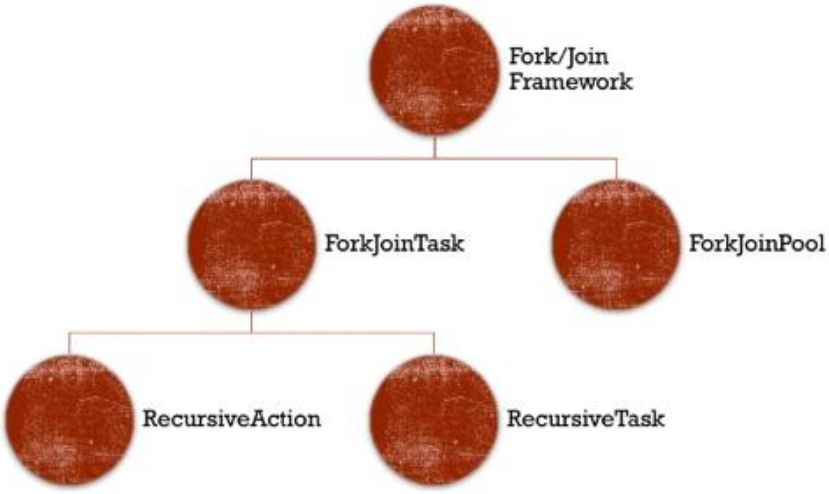
When a task has split itself up into subtasks, the task waits until the subtasks have finished executing. Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

10.16 The ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. The ForkJoinPool located in the java.util.concurrent package, so the full class name is java.util.concurrent.ForkJoinPool.



Creating a ForkJoinPool

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

This example creates a ForkJoinPool with a parallelism level of 4. So generally, take the all core available.

```
int cores = Runtime.getRuntime().availableProcessors();
```

Submitting Tasks to the ForkJoinPool

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit **two types** of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction and RecursiveTask classes. How to use both of these tasks and how to submit them will be covered in the following sections.

RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction  {

    private long workLoad = 0;

    public MyRecursiveAction(long workLoad) {
        this.workLoad = workLoad;
    }

    @Override
    protected void compute() {

        //if work is above threshold, break tasks up into smaller tasks
        if(this.workLoad > 16) {
            System.out.println("Splitting workLoad : " + this.workLoad);

             List<MyRecursiveAction> subtasks =new ArrayList<>();

            subtasks.addAll(createSubtasks());

            for(RecursiveAction subtask : subtasks){
                subtask.fork();
            }

        } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
```

```
        }
    }

    private List<MyRecursiveAction> createSubtasks() {
        List<MyRecursiveAction> subtasks = new ArrayList<>();

       MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);
       MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }

}
```

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.

You can schedule a MyRecursiveAction for execution like this:

```
MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);
```

RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;


public class MyRecursiveTask extends RecursiveTask<Long> {

   private long workLoad = 0;

   public MyRecursiveTask(long workLoad) {
      this.workLoad = workLoad;
   }

   protected Long compute() {

      //if work is above threshold, break tasks up into smaller tasks
      if(this.workLoad > 16) {
         System.out.println("Splitting workLoad : " + this.workLoad);

         List<MyRecursiveTask> subtasks =new ArrayList<>();
```

```
            subtasks.addAll(createSubtasks());

            for(MyRecursiveTask subtask : subtasks){
               subtask.fork();
            }

            long result = 0;
            for(MyRecursiveTask subtask : subtasks) {
               result += subtask.join();
            }
            return result;

         } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
            return workLoad * 3;
         }
      }

      private List<MyRecursiveTask> createSubtasks() {
         List<MyRecursiveTask> subtasks =
         new ArrayList<MyRecursiveTask>();

         MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);
         MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

         subtasks.add(subtask1);
         subtasks.add(subtask2);

         return subtasks;
      }
   }
```

This example is like the RecursiveAction example except it returns a result. The class MyRecursiveTask extends RecursiveTask<Long> which means that the result returned from the task is a Long.

The MyRecursiveTask example also breaks the work down into subtasks, and schedules these subtasks for execution using their fork() method.

- *Note: ForkJoinPool Javadoc states that the number of threads must be **a power of two**.*

Additionally, this example then receives the result returned by each subtask by calling the join() method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / merging of subtask results may occur recursively for several levels of recursion.

You can schedule a RecursiveTask like this:

```
MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);
```

Notice how you get the result out from the ForkJoinPool.invoke() method call.

S. No.	wait()	notify()
1.	When wait() is called on a thread holding the monitor lock, it surrenders the monitor lock and enters the waiting state.	When the notify() is called on a thread holding the monitor lock, it symbolizes that the thread is soon going to surrender the lock.
2.	There can be multiple threads in the waiting state at a time.	One of the waiting threads is randomly selected and notified about the same. The notified thread then exits the waiting state and enters the blocked state where it waits till the previous thread has given up the lock and this thread has acquired it. Once it acquires the lock, it enters the runnable state where it waits for CPU time and then it starts running.
3.	The wait() method is used for interthread communication.	The notify() method is used to wake up a single thread
4.	The wait() method is a part of java.lang.Object class	The notify() method does not have any return type value
5.	The wait() method is tightly integrated with the synchronization lock	The notify() method is used to give the notification for one thread for a particular object

11. Exception Handling:

In Java, an exception is an event that occurs during the execution of a program and disrupts the normal flow of instructions. When an exceptional event occurs, an object representing the exception is created and thrown in the method that caused the error.

Type of exception:

1) **Throwable**: The root class for all exceptions in Java.
2) **Exception**: A subclass of Throwable used to represent exceptional conditions that can be caught and handled.
3) **Error**: A subclass of Throwable used to represent severe, typically unrecoverable errors in the JVM.

12. Optional Class

It is used to represent optional values that is either exist or not exist. Optional is a container that either contains a non-null value or nothing (empty Optional).

One of the key benefits of using Optional is that it forces you to handle the case where the value is absent.

Main Advantage of Optional is:

- It is used to avoid null checks.
- It is used to avoid “NullPointerException”.

Example:

```
Student student = getByName("test");
```

Student may be null so to handle or avoid NPE, we do the following.

```
Optional<Student> student= Optional.ofNullable(getByName("Test"));
```

If u need some default value if null then

```
Student student =Optional.ofNullable(getByName("test")).orElse(new Student("Default"));
System.out.println(student.getName());
```

If u need to throw exception when value is null then

```
Student  student = Optional.ofNullable(getStudentWithName("fs")).orElseThrow(()-> new
StudentNotFoundException("the Student is not Present "));
```

13. Reflections classes

In Java, reflection is a feature that lets you inspect and manipulate classes, methods, fields, and constructors at runtime — even if you don't know their names at compile time.

Class / Interface	Purpose
Class (in java.lang)	Entry point for reflection — represents metadata about a loaded class or interface.
Field	Represents a class field (member variable). Allows reading/writing its value.
Method	Represents a method. Allows invocation at runtime.
Constructor	Represents a constructor. Allows creation of new instances.
Parameter	Represents a method/constructor parameter.
Modifier	Utility class for checking modifiers (public, private, static, etc.).
Array	Provides static methods to dynamically create and access arrays.

Accessing private members using reflection

The following code shows how to access private member variables and methods in Java.

Foo class

```
@SuppressWarnings("unused")
public class Foo {
  private int a = 10;

  private void m() {
      System.out.println("this is private method from class Foo");
  }
```

Bar class

```
class Bar {
```

```
  public static void main(String[] args) throws ReflectiveOperationException {
    Foo foo = Foo.class.newInstance();
    Class<Foo> clazz = Foo.class;
    Method[] methods = clazz.getDeclaredMethods();
    for (int i = 0; i < methods.length; i++) {
      methods[i].setAccessible(true);
      methods[i].invoke(foo);
    }

    Field[] fields = clazz.getDeclaredFields();

    for (int i = 0; i < fields.length; i++) {
      fields[i].setAccessible(true);
      fields[i].setInt(foo, 1);
      System.out.println(fields[i].getInt(foo));
    }
  }
}
```

Now to restrict it below two ways

1) Adding code in private method

```
public class Foo {
  private int a = 10;

  private void m() {
      /**One way to restrict access checks [Only Applicable for methods] */
      ReflectPermission perm = new ReflectPermission ("suppressAccessChecks", "");
      AccessController.checkPermission(perm);
      System.out.println("this is private method from class Foo");
  }
```

2) Adding custom security manager code

```
public class Foo {
  static {
    try {
      System.setSecurityManager(new MySecurityManager());
    } catch (SecurityException se) {
      System.out.println("SecurityManager already set!");
    }
  }

  private int a = 10;

  private void m() {
    System.out.println("this is private method from class Foo");
  }
   }

class MySecurityManager extends SecurityManager {
  @Override
    public void checkPermission(Permission perm) {
```

```
    if (perm.getName().equals("suppressAccessChecks"))
      throw new SecurityException("Can not change the permission.");
  }
}
```

14. Java 8's New features:

14.1 Date and Time API over Old Date API and Joda Time API

- Most of the API is deprecated.
- Less Readability.
- java.util.Date is Mutable and not Thread-Safe.
- java.text.SimpleDateFormat is not Thread-Safe.
- Less Performance.

14.2 Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

There are following types of method references in java:

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.

14.3 Functional Interface

An Interface that contains exactly one abstract method is known as **functional interface**. It can have any number of defaults, static methods but can contain only one abstract method. It can also declare methods of object class. Functional Interface is also known as Single Abstract Method Interfaces or **SAM** Interfaces.

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
final Supplier<Employee> supplier = () -> new Employee("empId","","");
```

Consumer is a functional interface. Like Supplier, it has one abstract functional method accept(T t)and a default method andThen(Consumer<? super T> after)

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

15. Heap Dump:

Differences Between Heap Dump, Thread Dump and Core Dump:

Heap dump contains a saved copy of the current state of all objects in use at runtime.
Heap Dump Format-

- Classic
- Portable

The heap dump can be in **two** formats – the **classic** format and the **Portable** Heap Format (PHD).
The classic format is human-readable, while the Portable is in binary and needs tools for further analysis.
PHD is the default for a heap dump.

Generate Heap Dump-

```
jmap -dump:live,format=b,file=heapdump.hprof <PID>
```

Note: it will generate the dump for live (reachable) objects only.

Auto Generate OutOfMemory Heap Dump-

Ensure the JVM is started with `-XX:+HeapDumpOnOutOfMemoryError` to auto-generate on OOM.

The **thread dump** contains the snapshot of all threads in a running Java program at a specific instant. It can help detect threads stuck in an infinite loop. It can also help identify deadlocks, where multiple threads are waiting for one other to release resources.

Generate Thread Dump- java Stack Tool (`jstack`) is used to generate Thread Dump.

```
jstack -l <PID> > threaddump.txt
```

l – includes locked monitors and synchronizers

Core Dump: A low-level **binary snapshot of the entire process memory**, including native memory, heap, threads, and registers.

Generate Core Dump-
Gcore (Linux only) is used to generate core dump.

```
gcore -o /tmp/core_dump <PID>
```

Comparison table-

Feature	Heap Dump	Thread Dump	Core Dump
Scope	Java heap (objects only)	All Java threads (stack trace)	Entire process memory
Usage	Memory leaks, GC issues	Deadlocks, CPU spikes	JVM/native crashes
Format	.hprof binary	Text (thread states)	OS binary (core image)
Tool Examples	jmap, MAT	jstack, VisualVM	gdb, jhsdb, crash
Size	Medium	Small	Large
Language Scope	Java only	Java only	Java + native (C/C++, JNI)

16. Java Profiler – Overview & Tools

A **Java Profiler** is a tool used to **monitor**, **analyze**, and **optimize** the performance of Java applications at runtime.
Profilers help you identify:

- CPU hotspots (methods taking most time)

- Memory leaks or high memory usage
- Thread contention or deadlocks
- GC behavior and performance bottlenecks

Below are few Popular Java Profilers-

Profiler	Type	Features	Best For
VisualVM	Free, GUI	CPU, Memory, GC, Thread profiling; Heap/thread dumps	General-purpose, lightweight
JFR (Java Flight Recorder)	Built-in (Java 11+)	Low-overhead production profiling	Long-running production apps
JMC (Java Mission Control)	GUI for JFR	Analyze JFR recordings	Deep JVM-level profiling
YourKit	Commercial	CPU, memory, threads, SQL/JDBC, async	In-depth professional analysis
JProfiler	Commercial	CPU, memory, thread profiling, DB, HTTP, I/O	Advanced, UI-rich profiler
Async Profiler	Open Source, CLI/GUI	Low-overhead, native stack traces, flame graphs	High-performance profiling
Eclipse MAT	Free, GUI	Heap dump analysis, memory leaks	Offline memory leak detection

Use Case	Tool Suggestion
Debugging high CPU usage	VisualVM, Async Profiler, JFR
Investigating memory leaks	Eclipse MAT, JProfiler, VisualVM
Thread contention/deadlocks	JFR, VisualVM, JProfiler
Production-safe profiling	Java Flight Recorder, Async Profiler
Heap dump analysis	Eclipse MAT, VisualVM
Web app profiling (Servlets, DB)	JProfiler, YourKit

17. Java 17's New Features

New Feature adds comparison between java 8, 11 and 17

Feature / Criteria	Java 8 (2014)	Java 11 (2018)	Java 17 (2021)
New Language Features	✓ Lambdas ✓ Streams API ✓ Optional	✓ var (local inference) ✓ New HTTP Client	✓ Sealed Classes ✓ Switch Expressions ✓ Text Blocks ✓ Records
Garbage Collection (GC)	Parallel GC (default)	G1 GC (default)	G1 (default), ZGC, Shenandoah
Performance Improvements	Baseline	Faster startup & lower memory	Improved GC, faster JIT, better memory efficiency
New APIs	Date-Time (java.time), Nashorn	HTTP/2 Client API	Foreign Function & Memory API (preview)
JVM Features	Classic HotSpot	Class Data Sharing (CDS)	Strong encapsulation (no --illegal-access)
Deployment Impact	Widely adopted, old apps	Modern container use begins	Recommended for new projects
Tooling	javac, jvisualvm	jlink (modular runtime)	jpackage (native packaging)

a) Sealed Classes

- Restrict which classes can extend or implement a superclass.
- Improves control over inheritance and enables better code modelling.

```
public sealed class Shape permits Circle, Square {}
final class Circle extends Shape {}
final class Square extends Shape {}
//No Other class can extend Shape
```

b) Pattern Matching

Simplifies casting after an instanceof check

```
if (obj instanceof String s) {
   System.out.println(s.toLowerCase()); // no explicit cast
}
```

c) Switch Expressions

Java 17 introduced **enhanced switch expressions** as a **standard (non-preview)** feature. These make switch statements:

- More **concise**
- **Safer** (exhaustiveness checked at compile time for enums/sealed types)
- **Expression-based** (can return a value)
- Support **pattern matching** (future expansion)

d) Records

Records are a special kind of class in Java introduced to reduce boilerplate code when creating data-carrying classes (like DTOs, value objects, etc.)

```
public record Person(String name, int age) {}
```

This automatically equivalent to:

- private final fields
- A constructor
- getters() with names as fields
- equals(), hashCode(), and toString()

```
class Main {
        public record Person(String name, int age) {}

        public static void main(String[] args) {
        Person p = new Person("Alice", 30);
            System.out.println(p.name());   // Alice
            System.out.println(p.age());    // 30
            System.out.println(p); //Person [name=Alice, age=30]
    }
}
```

Rules for Records

1. **Implicitly final** – cannot be extended.
2. **Cannot define mutable fields** (no setters).
3. Can implement interfaces.
4. Can add custom methods or constructors.

e) Text Blocks

Simplifies multiline strings without needing lots of \n and quotes.

Example:

```
String json = """
   {
    "name": "John",
    "age": 30
   }
   """;
```

f) Enhanced switch Expressions

```
String result = switch (day) {
   case MONDAY -> "Start";
   case FRIDAY -> "End";
   default -> "Middle";
};
```

```java
//Java 8
switch (day) {
    case "MONDAY":
        result = "Start of week";
        break;
    case "FRIDAY":
        result = "End of week";
        break;
    default:
        result = "Midweek";
        break;
}
```

```java
//java 17
String result = switch (day) {
    case "MONDAY" -> "Start of week";
    case "FRIDAY" -> "End of week";
    default -> "Midweek";
};
```

g) Multi-label Case Support

```
String day = "SATURDAY";

String result = switch (day) {
   case "SATURDAY", "SUNDAY" -> "Weekend";
   case "MONDAY" -> "Start of week";
   default -> "Weekday";
};
```

h) Strong Encapsulation of Internal API

Many older Java libraries or applications used internal JDK APIs like `sun.misc.Unsafe`, `sun.reflect.Reflection`, etc. You can no longer access them without **explicit command-line flags** — this improves security and maintainability.

```
import sun.misc.Unsafe;  // Internal API
class Example {
   public static void main(String[] args) {
      Unsafe unsafe = Unsafe.getUnsafe();  // will Throws exception
   }
}
```

Exception in thread "main" java.lang.SecurityException: Unsafe

You can bypass it only during development or legacy migration using the --add-opens JVM argument:

--add-opens java.base/sun.misc=ALL-UNNAMED

i) New Garbage Collectors:

Epsilon Garbage Collector

Epsilon is a do-nothing (no-op) garbage collector that was released as part of JDK 11. It handles memory allocation but does not implement any actual memory reclamation mechanism. Once the available Java heap is exhausted, the JVM shuts down.

It can be used for ultra-latency-sensitive applications, where developers know the application memory footprint exactly, or even have (almost) completely garbage-free applications. Usage of the Epsilon GC in any other scenario is otherwise discouraged.

The JVM argument to use the Epsilon Garbage Collector is -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC.

Shenandoah

Shenandoah is a new GC that was released as part of JDK 12. Shenandoah's key advantage over G1 is that it does more of its garbage collection cycle work concurrently with the application threads. G1 can evacuate its heap regions only when the application is paused, while Shenandoah can relocate objects concurrently with the application.

Shenandoah can compact live objects, clean garbage, and release RAM back to the OS almost immediately after detecting free memory. Since all of this happens concurrently while the application is running, Shenandoah is more CPU intensive.

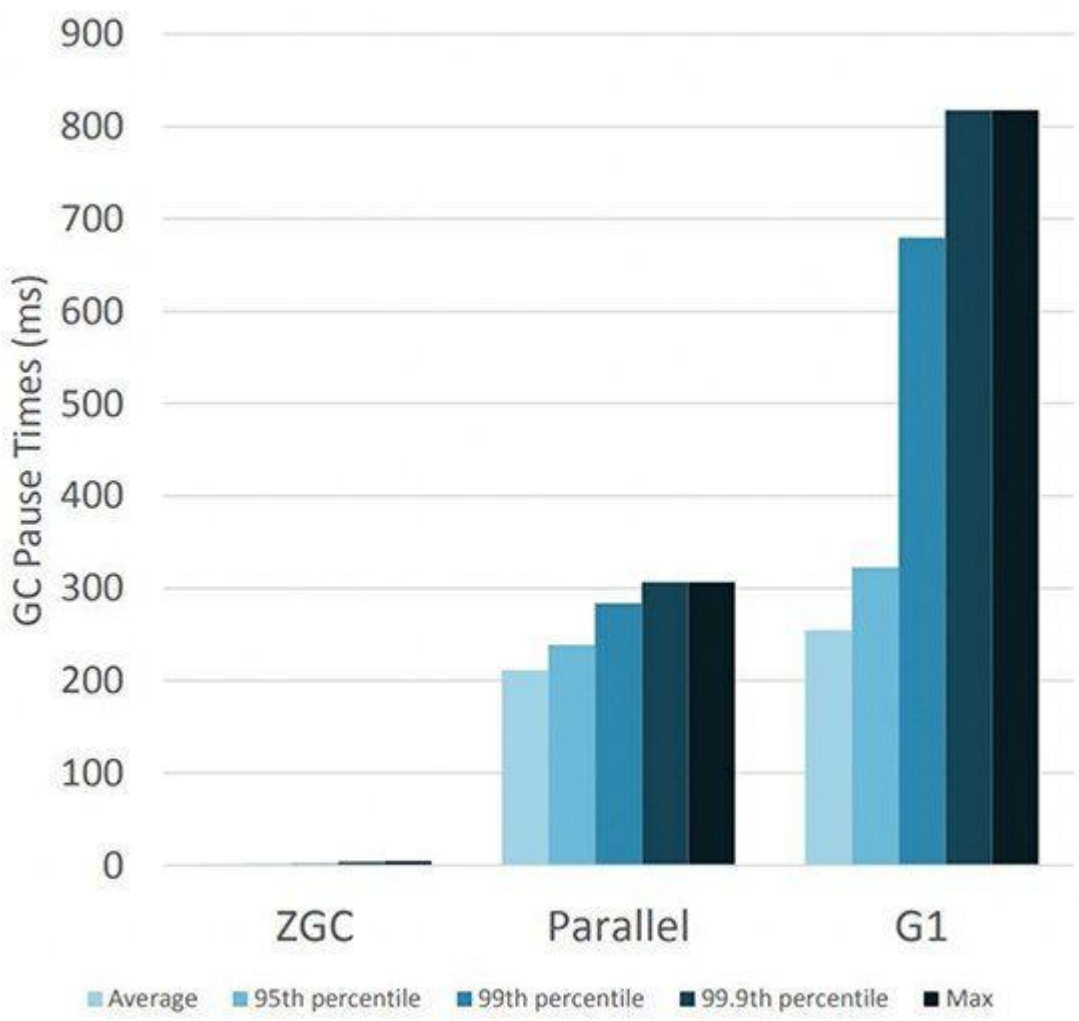
The JVM argument to use the Epsilon Garbage Collector is -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC.

ZGC

ZGC is another GC that was released as part of JDK 11 and has been improved in JDK 12. It is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes).

The primary goals of ZGC are low latency, scalability, and ease of use. To achieve this, ZGC allows a Java application to continue running while it performs all garbage collection operations. By default, ZGC uncommits unused memory and returns it to the operating system.

Thus, ZGC brings a significant improvement over other traditional GCs by providing extremely low pause times (typically within 2ms).



Source: oracle.com

The JVM argument to use the Epsilon Garbage Collector is -XX:+UnlockExperimentalVMOptions -XX:+UseZGC.

Note: Both Shenandoah and ZGC are planned to be made with production features and moved out of the experimental stage in JDK 15.

Thank You for Reading!

I hope this book has deepened your understanding of **Java core fundamentals** and given you the confidence to apply these concepts in real-world projects.
Java is not just a programming language — it's an evolving ecosystem. Mastering the basics and understanding the JVM's inner workings will empower you to write cleaner, faster, and more maintainable code.

As you continue your journey:

- Keep experimenting with code.
- Stay updated with Java's latest features.
- Never stop asking *why* something works the way it does.

After 18 years of working with Java, frameworks, and the JVM, I can assure you — the more you explore, the more fascinating it becomes.
If this book has helped you, I'd love to hear your feedback. Your thoughts will guide future editions and help me create resources that matter most to developers like you.

Happy Coding and see you in the next project!

— Neeraj Sachan
Technical Architect & Java Enthusiast
Email: neerajsachan2006@gmail.com