**OOPs** (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts: **Inheritance, Polymorphism, Abstraction, Encapsulation**

Apart from these concepts, there are some other terms which are used in Object-Oriented design: **Coupling, Cohesion, Association, Aggregation, Composition**  
  
**Object**: Any entity that has state and behavior is known as an object. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

**Class**: Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.  
  
**Inheritance**: When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.  
  
**Polymorphism**: If one task is performed in different ways, it is known as polymorphism. In Java, we use method overloading and method overriding to achieve polymorphism.  
  
**Abstraction**: Hiding internal details and showing functionality is known as abstraction. In Java, we use abstract class and interface to achieve abstraction.  
  
**Encapsulation**: Binding (or wrapping) code and data together into a single unit are known as encapsulation. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.  
  
**Coupling**: Coupling refers to the knowledge or information or dependency of another class. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can **use interfaces for the weaker coupling** because there is no concrete implementation.

**Cohesion**: Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.  
  
**Cohesion** is an indication of how related and focused the responsibilities of an software element are.  
**Coupling** refers to how strongly a software element is connected to other elements.  
The software element could be class, package, component, subsystem or a system. And while designing the systems it is recommended to have software elements that have **High cohesion** and support **Low coupling**.  
  
**Association**: Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects: ***One to One, One to Many, Many to One, Many to Many*.** Association can be unidirectional or bidirectional.

**Aggregation**: Aggregation is **a way to achieve Association**. Aggregation represents the relationship where one object contains other objects as a part of its state. ***It represents the weak relationship between objects***. It is also termed as a **has-a** relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

**Composition**: The composition is also a **way to achieve Association**. The composition represents the relationship where one object contains other objects as a part of its state. ***There is a strong relationship between the containing object and the dependent object***. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.  
  
**General contract associated with hashCode() method**

The**hashCode()** method should return the same integer value for the same object for each calling of this method unless the value stored in the object is modified.

If two objects are equal (according to **equals()** method) then the **hashCode()** method should return the same integer value for both the objects.

But, it is not necessary that the**hashCode()** method will return the distinct result for the objects that are not equal (according to **equals()** method.

**What is Java Architecture?**

* In Java, there is a process of compilation and interpretation.
* The code written in [Java](https://www.edureka.co/blog/java-tutorial/), is converted into byte codes which is done by the Java Compiler.
* The byte codes, then are converted into machine code by the JVM.
* The Machine code is executed directly by the machine.

This diagram illustrates the internal working of a Java code, or precisely, Java Architecture!  


**Components of Java Architecture** There are three main components of Java language: JVM, JRE, and JDK.  
 **Differences between JDK, JRE and JVM.**

* **JDK** – **Java Development Kit** which provides the environment to **develop and execute (run)** the Java program. JDK is a kit (or package) which includes two things
  + 1. Development Tools (to provide an environment to develop your java programs)
    2. JRE (to execute your java program).

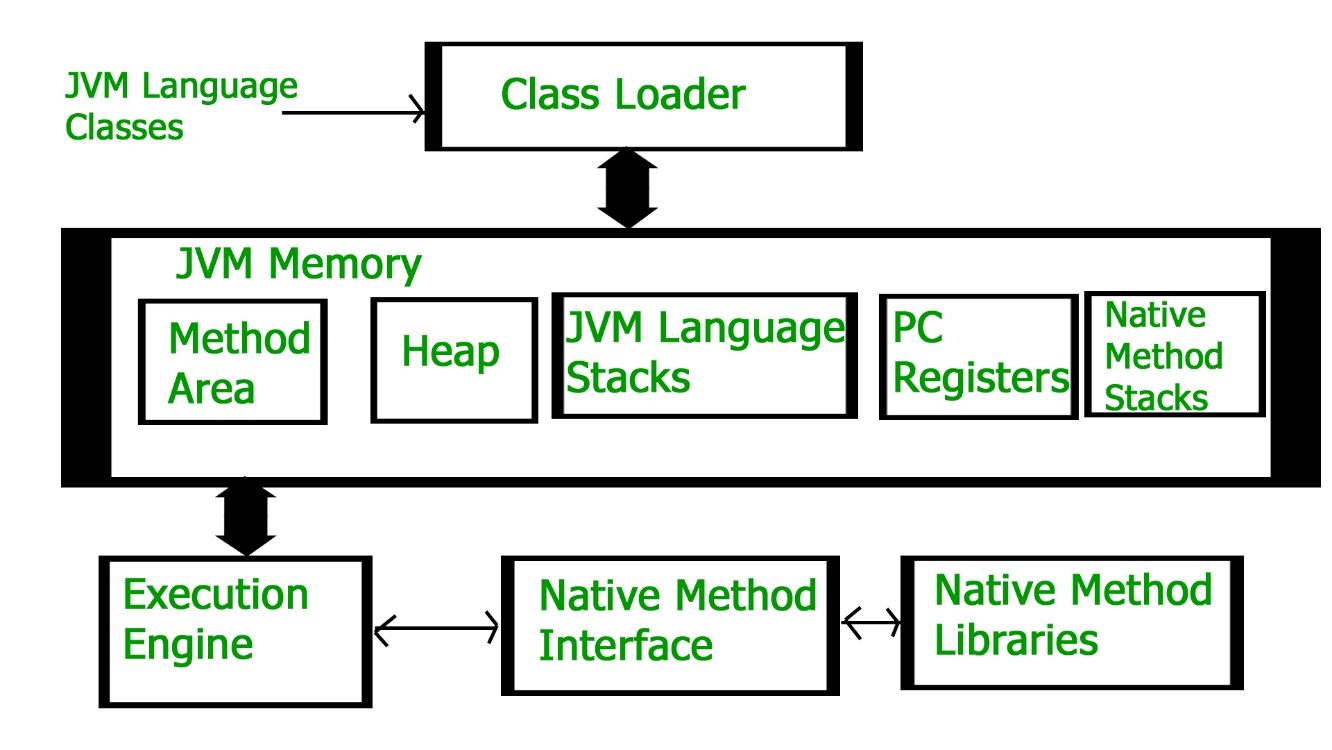
**Note :** JDK is only used by Java Developers.

* **JRE** – **Java Runtime Environment** is an installation package which provides environment to **only run (not develop)** the java program (or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* **JVM** – **Java Virtual machine** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

**How does JRE works?**  
To understand how the JRE works let us consider a Java source file saved as *Example.java*. The file is compiled into a set of Byte Code that is stored in a “*.class*” file. Here it will be “*Example.class*“.  
  
  
  
The following diagram depicts what is done at compile time.  
  
The following actions occur at runtime.

* **Class Loader** loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.
* **Byte Code Verifier** The JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.  
    
  The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.  
    
  
* **Interpreter** at runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:
  + Execute the Byte Code
  + Make appropriate calls to the underlying hardware

Both operations can be shown as:  
  
  
  
To understand the interactions between JDK and JRE consider the following diagram.  
  


**How JVM Works – JVM Architecture?**JVM acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE.  
  
When we compile a .java file, .class files (contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.  
  
  
  


**Class Loader Subsystem** It is mainly responsible for three activities. ***Loading, Linking, Initialization***  
  
**Loading**: The Class loader reads the “.class” file, generate the corresponding binary data and save it in the method area. For each “.class” file.  
  
**Linking**: Performs verification, preparation, and (optionally) resolution.

1. **Verification**: It ensures the correctness of the .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.
2. **Preparation**: JVM allocates memory for class variables and initializing the memory to default values.
3. **Resolution**: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

**Initialization**: In this phase, all static variables are assigned with their values defined in the code and static block (if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.   
  
There are three class loaders: ***Bootstrap, Extension, System***

1. **Bootstrap** class loader: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in the “JAVA\_HOME/jre/lib” directory. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.
2. **Extension** class loader: It is a child of the bootstrap class loader. It loads the classes present in the extensions directories “JAVA\_HOME/jre/lib/ext”(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.
3. **System/Application** class loader: It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

**Note**: ***JVM follows the Delegation-Hierarchy principle to load classes***. System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader. If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. At last, if the system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.  


 **1) ClassLoader** The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

**2) Method Area** JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

**3) Heap** All the [Objects](https://www.guru99.com/java-oops-class-objects.html), their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

**4) JVM language Stacks** Java language Stacks store local variables, and it’s partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

Stack Area: Stack Area generates when a thread creates. It can be of either fixed or dynamic size. The stack memory is allocated per thread. It is used to store data and partial results. It contains references to heap objects. It also holds the value itself rather than a reference to an object from the heap. The variables which are stored in the stack have certain visibility, called scope.

**Stack Frame:** Stack frame is a data structure that contains the thread?s data. Thread data represents the state of the thread in the current method.

* It is used to store partial results and data. It also performs dynamic linking, values return by methods and dispatch exceptions.
* When a method invokes, a new frame creates. It destroys the frame when the invocation of the method completes.
* Each frame contains own Local Variable Array (LVA), Operand Stack (OS), and Frame Data (FD).
* The sizes of LVA, OS, and FD determined at compile time.
* Only one frame (the frame for executing method) is active at any point in a given thread of control. This frame is called the current frame, and its method is known as the current method. The class of method is called the current class.
* The frame stops the current method, if its method invokes another method or if the method completes.
* The frame created by a thread is local to that thread and cannot be referenced by any other thread.

**5)  PC Registers** PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

**6) Native Method Stacks** Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

**7) Execution Engine** It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

**8) Native Method interface** The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

**9) Native Method Libraries** Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine. **JVM Memory Java (JVM) Memory Model**

1. ***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.
2. **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.
3. **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.
4. **PC Registers**: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **Native method stacks**: For **every thread, a separate native stack is created**. It stores native method information.

  
  
  
  
  
<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>



**Execution Engine** executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:  
 **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.  
 **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.  
 **Garbage Collector**: It destroys un-referenced objects.

**Software Code Compilation & Execution process** In order to write and execute a software program, you need the following  
**1) Editor**– To type your program into, a notepad could be used for this  
**2) Compiler**– To convert your high language program into native machine code  
**3) Linker**– To combine different program files reference in your main program together.  
**4) Loader**– To load the files from your secondary storage device like Hard Disk, Flash Drive, CD into RAM for execution. The loading is automatically done when you execute your code.  
**5) Execution** – Actual execution of the code which is handled by your OS & processor  
  
**What does Java Garbage Collector?**

JVM controls the garbage collector. JVM decides when to perform the garbage collection. We can also request to the JVM to run the garbage collector. But there is no guarantee under any conditions that the JVM will comply. JVM runs the garbage collector if it senses that memory is running low. When Java program request for the garbage collector, the JVM usually grants the request in short order. It does not make sure that the requests accept.

The point to understand is that "**when an object becomes eligible for garbage collection?**"

Every Java program has more than one thread. Each thread has its execution stack. There is a thread to run in Java program that is a main() method. Now we can say that an object is eligible for garbage collection when no live thread can access it. The garbage collector considers that object as eligible for deletion. If a program has a reference variable that refers to an object, that reference variable available to live thread, this object is called **reachable**.

Here a question arises that "**Can a Java application run out of memory?**"  
The answer is yes. The garbage collection system attempts to objects from the memory when they are not in use. Though, if you are maintaining many live objects, garbage collection does not guarantee that there is enough memory. Only available memory will be managed effectively.

### **Types of Garbage Collection** There are five types of garbage collection are as follows:

* **Serial GC:** It uses the mark and sweeps approach for young and old generations, which is minor and major GC.
* **Parallel GC:** It is similar to serial GC except that, it spawns N (the number of CPU cores in the system) threads for young generation garbage collection.
* **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.

## Mark and Sweep Algorithm

JRockit JVM uses the mark, and sweep algorithm for performing the garbage collection. It contains two phases, the mark phase, and the sweep phase.

**Mark Phase:** Objects that are accessible from the threads, native handles, and other GC root sources are marked as live. Every object tree has more than one root objects. GC root is always reachable. So any object that has a garbage collection root at its root. It identifies and marks all objects that are in use, and the remaining can be considered garbage.



**Sweep Phase:** In this phase, the heap is traversed to find the gap between the live objects. These gaps are recorded in the free list and are available for new object allocation.

There are two improved versions of mark and sweep:

* **Concurrent Mark and Sweep**
* **Parallel Mark and Sweep**

### **Concurrent Mark and Sweep**

It allows the threads to continue running during a large portion of the garbage collection. There are following types of marking:

* **Initial marking:** It identifies the root set of live objects. It is done while threads are paused.
* **Concurrent marking:** In this marking, the reference from the root set are followed. It finds and marks the rest of the live objects in a heap. It is done while the thread is running.
* **Pre-cleaning marking:** It identifies the changes made by concurrent marking. Other live objects marked and found. It is done while the threads are running.
* **Final marking:** It identifies the changes made by pre-cleaning marking. Other live objects marked and found. It is done while threads are paused.

### **Parallel Mark and Sweep**

It uses all available CPU in the system for performing the garbage collection as fast as possible. It is also called the parallel garbage collector. Threads do not execute when the parallel garbage collection executes.

**Pros of Mark and Sweep**

* It is a recurring process.
* It is an infinite loop.
* No additional overheads allowed during the execution of an algorithm.

**Cons of Mark and Sweep**

* It stops the normal program execution while the garbage collection algorithm runs.
* It runs multiple times on a program.

<https://www.studytonight.com/java/string-handling-in-java.php> **String** is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.



In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.



### **What is an Immutable object?**

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

<https://www.studytonight.com/java/multithreading-in-java.php>  
  
**Multithreading in Java**  
Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.  
  
Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.  
  
In Java, the word **thread** means two different things.

* An instance of **Thread** class.
* or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.  
  
  
  
**Advantage of Multithreading**Multithreading **reduces** the CPU **idle time** that increase overall performance of the system. Since thread is lightweight process then it takes **less memory** and perform **context switching** as well that helps to share the memory and reduce time of switching between threads.  
  
**Multitasking**Multitasking is a process of performing multiple tasks simultaneously. We can understand it by computer system that perform multiple tasks like: writing data to a file, playing music, downloading file from remote server at the same time.  
  
Multitasking can be achieved either by using multiprocessing or multithreading. Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.  
  
**Why Multithreading ?**Thread has many advantages over the process to perform multitasking. Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system. To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently. So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.

### **How to Create Thread ?** To create a thread, Java provides a class **Thread** and an interface **Runnable** both are located into java.lang package. We can create thread either by extending Thread class or implementing Runnable interface. Both includes a run method that must be override to provide thread implementation. It is recommended to use Runnable interface if you just want to create a thread but can use Thread class for implementation of other thread functionalities as well. **Life cycle of a Thread** Like process, thread have its life cycle that includes various phases like: new, running, terminated etc. we have described it using the below image.



1. **New :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable :** After invocation of start() method on new thread, the thread becomes runnable.
3. **Running :** A thread is in running state if the thread scheduler has selected it.
4. **Waiting :** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task.

**Daemon Thread**Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. ***Garbage collection thread is one of the system generated daemon thread that runs in background***. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.

**Thread Pool**  
In Java, is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.

**Thread Priorities**  
In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The ***range of the priority is between 1 to 10*** and there are three variables which are static to define priority in a Thread Class.

**Note:**Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

# **Java Thread Class**

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.  
It provides constructors and methods to support multithreading. It extends object class and implements Runnable interface.  
 **Signature of Thread class**

public class Thread extends Object implements Runnable**Thread Class Priority Constants**

|  |  |
| --- | --- |
| **Field** | **Description** |
| MAX\_PRIORITY | It represents the maximum priority that a thread can have. |
| MIN\_PRIORITY | It represents the minimum priority that a thread can have. |
| NORM\_PRIORITY | It represents the default priority that a thread can have. |

Constructors of Thread class

1. **Thread**()
2. **Thread**(String str)
3. **Thread**(Runnable r)
4. **Thread**(Runnable r, String str)
5. **Thread**(**ThreadGroup** group, **Runnable** target)
6. **Thread**(**ThreadGroup** group, **Runnable** target, String name)
7. **Thread**(**ThreadGroup** group, **Runnable** target, String name, long stackSize)
8. **Thread**(**ThreadGroup** group, **String** name)

### **Thread Class Methods** Thread class also defines many methods for managing threads. Some of them are,

|  |  |
| --- | --- |
| **Method** | **Description** |
| setName() | to give thread a name |
| getName() | return thread's name |
| getPriority() | return thread's priority |
| isAlive() | checks if thread is still running or not |
| join() | Wait for a thread to end |
| run() | Entry point for a thread |
| sleep() | suspend thread for a specified time |
| start() | start a thread by calling run() method |
| activeCount() | Returns an estimate of the number of active threads in the current thread's thread group and its subgroups. |
| checkAccess() | Determines if the currently running thread has permission to modify this thread. |
| currentThread() | Returns a reference to the currently executing thread object. |
| dumpStack() | Prints a stack trace of the current thread to the standard error stream. |
| getId() | Returns the identifier of this Thread. |
| getState() | Returns the state of this thread. |
| getThreadGroup() | Returns the thread group to which this thread belongs. |
| interrupt() | Interrupts this thread. |
| interrupted() | Tests whether the current thread has been interrupted. |
| isAlive() | Tests if this thread is alive. |
| isDaemon() | Tests if this thread is a daemon thread. |
| isInterrupted() | Tests whether this thread has been interrupted. |
| setDaemon(boolean on) | Marks this thread as either a daemon thread or a user thread. |
| setPriority(int newPriority) | Changes the priority of this thread. |
| yield() | A hint to the scheduler that the current thread is willing to yield its current use of a processor. |

### **Some Important points to Remember**

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

**Runnable Interface**It also used to create thread and should be used if you are only planning to override the run() method and no other Thread methods.  
  
**Signature**@FunctionalInterface   
public interface Runnablenable Interface **Method**

It provides only single method that must be implemented by the class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| run() | It runs the implemented thread. |

**Shutdown hook**In Java, Shutdown hook is used to clean-up all the resource, it means closing all the files, sending alerts etc. We can also save the state when the JVM shuts down. Shutdown hook mostly used when any code is to be executed before any JVM shuts down. Following are some of the reasons when the JVM shut down:

* Pressing ctrl+c on the command prompt
* When the System.exit(int) method is invoked.
* When user logoff or shutdown etc

#### **addShutdownHook(Thread hook)**

The addShutdownHook(Thread hook) method is used to register the thread with the virtual machine. This method is of Runtime class.

class Demo6 extends Thread{   
 public void run(){   
 System.out.println("Shutdown hook task is Now completed...");   
 }   
}

public class ShutdownDemo1{   
 public static void main(String[] args)throws Exception {   
 Runtime obj=Runtime.getRuntime();   
 obj.addShutdownHook(new Demo6());   
 System.out.println("Now main method is sleeping... For Exit press ctrl+c");   
 try{  
 Thread.sleep(4000);  
 }catch (Exception e) {}   
 }   
}



### **OutOfMemory Exception** In Java, as we know that all objects are stored in the heap. The objects are created using the new keyword. The OutOfMemoryError occurs as follow:

out-of-memory-exception

This error occurs when Java Virtual Machine is not able to allocate the object because it is out of memory and no memory can be available by the garbage collector.  
  
The meaning of OutOfMemoryError is that something wrong is in the program. Many times the problem can be out of control when the third party library caches strings.  
  
**Basic program in which OutOfMemoryError can occur  
  
Example:**

import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
public class OutOfMemoryDemo1 {  
 public static void main(String[] args) {  
 Listobj = new ArrayList<>();  
 Random obj1= new Random();  
 while (true)  
 obj.add(obj1.nextInt());  
 }  
}



### **Program in which OutOfMemoryError can occur because of low memory**

**Example:**

public class OutOfMemoryErrorDemo2{  
 public static void main(String[] args) {  
 Integer[] a = new Integer[100000\*10000\*1000];  
 System.out.println("Done");  
 }  
}

# out-of-memory-exception-example-2.JPG **Creating a thread in Java**

To implement multithreading, Java defines two ways by which a thread can be created.

* By implementing the **Runnable** interface.
* By extending the **Thread** class.

### **What if we call run() method directly without using start() method?** In above program if we directly call run() method, without using start() method,

public static void main(String args[]){  
 MyThread mt = new MyThread();  
 mt.run();  
}

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.  
  


### **Can we Start a thread twice?** No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

### **Thread Pool** In Java, is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.



### **There are four methods of a Thread pool. They are as following:**

1. newFixedThreadPool(int)  
2. newCachedThreadPool()  
3. newSingleThreadExecutor()  
4. newScheduledThreadPool()

### **Following are the steps for creating a program of the thread pool** 1. create a runnable object to execute. 2. using executors create an executor pool 3. Now Pass the object to the executor pool 4. At last shutdown the executor pool. Example:

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
class WorkerThread implements Runnable{  
 private String message;  
 public WorkerThread(String a){  
 this.message=a;  
 }

public void run(){  
 System.out.println(Thread.currentThread().getName()+"   
 (Start) message = "+message);  
 processmessage();  
 System.out.println(Thread.currentThread().getName()+" (End)");  
 }

private void processmessage(){  
 try{   
 Thread.sleep(5000);   
 }catch (InterruptedException e){  
 System.out.println(e);   
} }

}

public class ThreadPoolDemo1{   
 public static void main(String[] args){  
 ExecutorService executor = Executors.newFixedThreadPool(5);  
 for (int i = 0; i < 10; i++){  
 Runnable obj = new WorkerThread("" + i);  
 executor.execute(obj);  
 }  
 executor.shutdown();  
 while (!executor.isTerminated()){  
 }  
 System.out.println("\*\*\*\*\*\*\*\*All threads are Finished\*\*\*\*\*\*\*\*");  
 }  
}

# **Joining threads in Java**

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.  
  
The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

*final* boolean **isAlive()**

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

*final* void **join()** throws **InterruptedException**

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

*final* void **join**(long *milliseconds*) throws **InterruptedException**

As we have seen in the [Introduction to MultiThreading](http://www.studytonight.com/java/multithreading-in-java), the main thread must always be the last thread to finish its execution. Therefore, we can use Thread join() method to ensure that all the threads created by the program has been terminated before the execution of the main thread.

# **Java Sleeping Thread** To sleep a thread for a specified time, Java provides sleep method which is defined in Thread class. The sleep method is an overloaded method which are given below. It throws interrupted exception so make sure to provide proper handler.

It always pause the current thread execution. Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.  
  
**Syntax**

sleep(long millis)throws InterruptedException  
sleep(long millis, int nanos)throws InterruptedException

# **Thread.sleep()** interacts with the thread scheduler to put the current thread in wait state for specified period of time. Once the wait time is over, thread state is changed to runnable state and wait for the CPU for further execution. So the actual time that current thread sleep depends on the thread scheduler that is part of operating system. **Java Naming Thread**

Each thread in Java has its own name which is set by the JVM by default. Although there are many other attributes associated to the thread like: id, priority etc.

# we can get name of a thread by calling getName() method of Thread class. If we wish to set new name of the thread then **setName()** method can be used. Both methods belong to Thread class and even we can set name of thread by passing into constructor during creating object. **Java Thread Priorities**

Priority of a thread describes how early it gets execution and selected by the thread scheduler. In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three constant variables which are static and used to fetch priority of a Thread. They are as following:

# 1. public static int **MIN\_PRIORITY** It holds the minimum priority that can be given to a thread. The value for this is 1. 2. public static int **NORM\_PRIORITY** It is the default priority that is given to a thread if it is not defined. The value for this is 0. 3. public static int **MAX\_PRIORITY** It is the maximum priority that can be given to a thread. The value for this is 10. Get and Set methods in Thread priority 1. **public final intgetPriority()** In Java, getPriority() method is in java.lang.Thread package. it is used to get the priority of a thread. 2. **public final void setPriority(intnewPriority)** In Java setPriority(intnewPriority) method is in java.lang.Thread package. It is used to set the priority of a thread. The setPriority() method throws IllegalArgumentException if the value of new priority is above minimum and maximum limit. **Java Daemon Thread**

Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. Garbage collection thread is one of the system generated daemon thread that runs in background. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.



### **Following are the methods in Daemon Thread**

**1. void setDaemon(boolean status)**In Java, this method is used to create the current thread as a daemon thread or user thread. If there is a user thread as obj1 then obj1.setDaemon(true) will make it a Daemon thread and if there is a Daemon thread obj2 then calling obj2.setDaemon(false) will make it a user thread.  
  
**Syntax:**

public final void setDaemon(boolean on)

**2. boolean isDaemon()**In Java, this method is used to check whether the current thread is a daemon or not. It returns true if the thread is Daemon otherwise it returns false.  
**Syntax:**

public final booleanisDaemon()

### **Example:**

Lets create an example to create daemon and user threads. To create daemon thread setdaemon() method is used. It takes boolean value either true or false.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }   
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");   
  
 D1.setDaemon(true);   
 D1.start();   
 D2.start();   
 D3.setDaemon(true);   
 D3.start();   
 }   
}



### Example : Daemon thread Priority Since daemon threads are low level threads then lets check the priority of these threads. The priority we are getting is set by the JVM.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }  
 System.out.println(getName()+" priority "+Thread.currentThread().getPriority());  
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");

D1.setDaemon(true);   
 D1.start();   
 D2.start();   
 D3.setDaemon(true);   
 D3.start();   
 }   
}

D1 is Daemon thread  
D1 priority 5  
D2 is User thread  
D3 is Daemon thread  
D2 priority 5  
D3 priority 5  
  
**Example**While creating daemon thread make sure the setDaemon() is called before starting of the thread. Calling it after starting of thread will throw an exception and terminate the program execution.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }  
 System.out.println(getName()+" priority "+Thread.currentThread().getPriority());  
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");   
  
 D1.setDaemon(true);   
 D1.start();   
 D2.start();  
 D3.start();  
 D3.setDaemon(true);   
}   
}

D1 is Daemon threadException in thread "main"   
D1 priority 5  
D3 is User thread  
D2 is User thread  
D2 priority 5java.lang.IllegalThreadStateException  
D3 priority 5  
 at java.base/java.lang.Thread.setDaemon(Thread.java:1410)  
 at myjavaproject.DaemonDemo1.main(DaemonDemo1.java:32)

# **Java Synchronization**

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

### **Difference between synchronized keyword and synchronized block** When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution. synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

# **Which is more preferred - Synchronized method or Synchronized block?** In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only. **Java Interthread Communication**

Java provide benefits of avoiding thread pooling using inter-thread communication. The wait(), notify(), and notifyAll() methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context

* wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
* notify() wakes up a thread that called wait() on same object.
* notifyAll() wakes up all the thread that called wait() on same object.

## Difference between wait() and sleep()

### **Thread Pooling**

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| called from synchronised block | no such requirement |
| monitor is released | monitor is not released |
| gets awake when notify() or notifyAll() method is called. | does not get awake when notify() or notifyAll() method is called |
| not a static method | static method |
| wait() is generaly used on condition | sleep() method is simply used to put your thread on sleep. |

Pooling is usually implemented by loop i.e to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

## Thread Deadlock in Java



* Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.  
    
  https://www.geeksforgeeks.org/different-ways-create-objects-java/ **Different ways to create objects in Java**  
    
  1) **Using new Keyword** : Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).  
    
  2) **Using New Instance**: If we know the name of the class & if it has a public default constructor we can create an object –Class.forName. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn’t create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.  
    
  3) **Using clone() method**: Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
  To use clone() method on an object we need to implement Cloneable and define the clone() method in it.  
    
  4) **Using deserialization**: Whenever we serialize and then deserialize an object, JVM creates a separate object. In deserialization, JVM doesn’t use any constructor to create the object. To deserialize an object we need to implement the Serializable interface in the class.  
    
  5) **Using newInstance() method of Constructor class** : This is similar to the newInstance() method of a class. There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.  
    
  Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.  
    
  https://codippa.com/how-to-break-singleton-in-java/  
    
  **How to break Singleton class in java**  
  1. Break by Cloning : If a Singleton class implements java.lang.Cloneable interface then invoking clone() method on its single instance creates a duplicate object.  
    
  **Remedy**: Don’t make the class which should be Singleton implement java.lang.Cloneable. If it extends a class which implements Cloneable, then override clone method and throw CloneNotSupportedException from it. This will prevent clone creation.  
    
  2. Deserialization also breaks Singleton: Deserialization means bringing a saved object back to life. When a class is deserialized, a fresh instance of the class is created and its instance variables are then set to the values which were serialized.  
    
  **Remedy**: Override readResolve method and return the same Singleton instance every time. readResolve method is called after the object has been read from the stream but before it is returned to the calling code.   
    
  3. Reflection can instantiate a Singleton multiple times: Using java reflection api, we can tweak into a class by getting details like its fields, constructor, invoking its methods etc. Reflection can also be used to create new instance of a class. Obviously, if the class is Singleton, then creating a new instance again breaks its Singleton nature.  
    
  **Remedy**: Check the static instance to be not null and throw Error if it is not null which means one instance already exists as .  
    
    
    
  **Transaction Isolation Levels  
  Dirty Reads**: Transaction "A" writes a record. Meanwhile, Transaction "B" reads that same record before Transaction A commits. Later, Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.  
    
  **Non-Repeatable Reads**: Transaction "A" reads some record. Then Transaction "B" writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.  
    
  **Phantom Reads:**Transaction "A" reads a range of records. Meanwhile, Transaction "B" inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).  
    
  **DEFAULT:**Use the default isolation level of the underlying database.  
  **READ\_COMMITTED:**A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.  
  **READ\_UNCOMMITTED:**This isolation level states that a transaction may read data that is still uncommitted by other transactions.  
  **REPEATABLE\_READ:**A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.  
  **SERIALIZABLE:**A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.  
    
  <https://dzone.com/articles/spring-transaction-management>  
    
  **Transaction Propagation**

|  |  |
| --- | --- |
| **Propagation** | **Behavior** |
| **REQUIRED** | **Always executes in a transaction.** If there is an existing transaction, it uses it. If none exists, then only a new one is created. |
| **SUPPORTS** | **It may or may not run in a transaction.** If the current transaction exists, then it is supported. If none exists, then it gets executed without a transaction. |
| **NOT\_SUPPORTED** | **Always executes without a transaction.** If there is an existing transaction, it gets suspended. |
| **REQUIRES\_NEW** | **Always executes in a new transaction.** If there is an existing transaction, it gets suspended. |
| **NEVER** | **Always executes without any transaction.** It throws an exception if there is an existing transaction |
| **MANDATORY** | **Always executes in a transaction.** If there is an existing transaction, it is used. If there is no existing transaction, it will throw an exception. |

**How HashSet works in Java**When you create an object of HashSet in Java, it internally creates an instance of backup Map with default initial capacity 16 and default load factor 0.75 as shown below :  
  
/\*\* Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75). \*/

public **HashSet**() {

map **=** **new** **HashMap**<>();

}

Now let's see the code for add() and iterate() method from java.util.HashSet in Java to understand *how HashSet works internally in Java*.

**How Object is stored in HashSet?** As you can see below, a call to add(Object) is a delegate to put(Key, Value) internally, where Key is the object you have passed and value is another object,  called PRESENT, which is a constant in java.util.HashSet as shown below :

**private** transient **HashMap**<E, **Object**> map;

// Dummy value to associate with an Object in the backing Map

**private** static **final** **Object** **PRESENT** **=** **new** **Object**();

public boolean add(E e) {

**return** map.put(e, **PRESENT**)==**null**;

}

Since PRESENT is a constant, for all keys we have the same value in backup HashMap called map.

[](https://www.java67.com/2018/08/top-10-free-java-courses-for-beginners-experienced-developers.html)

**How Object is retrieved from HashSet** Now let's see the code for getting iterator for traversing over HashSet in Java. iterator() method from java.util.HashSet class returns iterator for backup Map **returned** by **map.keySet().iterator()** method.  
  
 /\*\*

\* Returns an iterator over the elements in this set. The elements are returned in no particular order.

\* @return an Iterator over the elements in this set

\* @see ConcurrentModificationException

\*/

public **Iterator**<E> iterator() { **return** map.keySet().iterator(); }

**How to use HashSet in Java - Example**

Using HashSet in Java is very simple, don't think it is Map but think more like Collection i.e. add elements by using add() method, check its return value to see if the object already existed in HashSet or not. Similarly use an iterator for retrieving elements from HashSet in Java.

## You can also use contains() method to check if an object already exists in HashSet or not. This method use [equals() method](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html) for comparing object for matching. You can also use the remove() method to remove objects from HashSet. Since the element of HashSet is used as a key in backup HashMap, they must implement the [equals() and hashCode()](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html) method. <https://www.java67.com/2016/01/difference-between-list-and-arraylist-variable-in-java.html#ixzz70loOwwDr> Why store ArrayList object on the List variable?

You might have seen something like this:  
List<Movie> listOfMovies = new ArrayList<Movie>()

The answer is to take advantage of Polymorphism. If you use interface than in the future if the new implementation is shipped, then you are not required to change your program.  
  
For example, an application written using **List** will work as expected whether you pass a [LinkedList](http://java67.blogspot.com/2016/01/how-to-implement-singly-linked-list-in-java-using-generics-example.html), [Vector](http://java67.blogspot.com/2012/09/arraylist-vs-vector-in-java-interview.html), or [ArrayList](http://java67.blogspot.com/2012/08/how-to-sort-arraylist-in-java-list.html) because they all implement List interface, they obey the contract exposed by the List interface  
  
  
  
  
  
**Difference between ArrayList and HashSet in Java**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **ArrayList** | **HashSet** |
| 1 | Implementation | ArrayList is the implementation of the list interface. | HashSet on the other hand is the implementation of a set interface. |
| 2 | Internal implementation | ArrayList internally implements Array for its implementation. | HashSet internally uses HashMap for its implementation. |
| 3 | Order of elements | ArrayList maintains the insertion order i.e., order of the object in which they are inserted. | HashSet is an unordered collection and doesn't maintain any order. |
| 4 | Duplicates | ArrayList allows duplicate values in its collection. | On other hand duplicate elements are not allowed in HashSet. |
| 5 | Index performance | ArrayList uses index for its performance i.e. its index based one can retrieve object by calling get(index) or remove objects by calling remove(index) | HashSet is completely based on object also it doesn't provide get() method. |
| 6 | Null Allowed | Any number of null value can be inserted in ArrayList without any restriction. | HashSet allows only one null value in its collection, after which no null value is allowed to be added. |

<https://beginnersbook.com/java-collections-tutorials>   
  
  
  
  
**Differences Between HashSet, LinkedHashSet and TreeSet In Java**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e. elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. Its performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparison of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

**Similarities Between HashSet, LinkedHashSet and TreeSet in Java:**

* All three doesn’t allow duplicate elements.
* All three are not synchronized.
* All three are Cloneable and Serializable.
* Iterator returned by all three is fail-fast in nature. i.e. You will get ConcurrentModificationException if they are modified after the creation of Iterator object.

|  |  |
| --- | --- |
|  |  |

**Differences between TreeMap, HashMap and LinkedHashMap in Java**<https://www.geeksforgeeks.org/differences-treemap-hashmap-linkedhashmap-java/>   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Difference Between Collections Vs Streams In Java**<https://javaconceptoftheday.com/collections-and-streams-in-java/>   
  
**1) Conceptual Difference**Collections are used to store and group the data in a particular data structure like **List**, **Set** or **Map**. But, streams are used to perform complex data processing operations like **filtering**, **matching**, **mapping** etc on stored data such as arrays, collections or I/O resources. That means, collections are mainly about data and streams are mainly about operations on data.  
  
**2) Data Modification**  
You can add to or remove elements from collections. But, you can’t add to or remove elements from streams. Stream consumes a source, performs operations on it and returns a result. They don’t modify even the source also.  
  
**3) External Iteration Vs Internal Iteration**The main specialty of Java 8 Streams is that you need not to worry about iteration while using streams. Streams perform iteration internally behind the scene for you. You just have to mention the operations to be performed on a source.  
  
**4) Traversal**  
Streams are traversable only once. If you traverse the stream once, it is said to be consumed. To traverse it again, you have to get new stream from the source again. But, collections can be traversed multiple times.  
  
**5) Eager Construction Vs Lazy Construction**  
Collections are eagerly constructed i.e. all the elements are computed at the beginning itself. But, streams are lazily constructed i.e intermediate operations are not evaluated until terminal operation is invoked.

|  |  |
| --- | --- |
| **Collections** | **Streams** |
| Collections are mainly used to store and group the data. | Streams are mainly used to perform operations on data. |
| You can add or remove elements from collections. | You can’t add or remove elements from streams. |
| Collections have to be iterated externally. | Streams are internally iterated. |
| Collections can be traversed multiple times. | Streams are traversable only once. |
| Collections are eagerly constructed. | Streams are lazily constructed. |
| Ex : List, Set, Map… | Ex : filtering, mapping, matching… |

 **Jdk 1.8 Cheat Sheet**  


**Docker**:  


**SQL**:

