

## **Day 10- Facilitation Guide**

### **(Threads, Synchronization, Inter-thread communication)**

#### **Index**

- I. Multitasking
- II. Threads
- III. Thread Scheduler
- IV. Synchronization
- V. Inter thread communication
- VI. JVM

#### **For (1.5 hrs) ILT**

**During the previous session we learnt about TreeMap, Comparator, Comparable and the importance of overriding equal() and hashCode() method, we covered a variety of important concepts:**

**TreeMap:** A TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

**Comparable Interface:** As the name suggests, Comparable is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class's "natural ordering".

**Comparator Interface:** The Comparator interface defines a compare(arg1, arg2) method with two arguments that represent compared objects, and works similarly to the Comparable.compareTo() method.

**Overriding the equals() and hashCode() methods:** Overriding the equals() and hashCode() methods is essential to ensure that custom classes work correctly when used as keys in hash-based collections and when you need to define logical equality for instances of your class.

**In this session, we will explore Thread and inter-thread communication along with all methods of the Thread class.**

Before delving into the concept of Threads, it's important to understand the necessity of threads and for that We need to understand Multitasking.

**Question:** With the rise of smartphones and social media, how has multitasking changed in the digital age?

## I. Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

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

### 1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- The process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the threads is low.

#### **What is better? Multiprocessing or multithreading?**

Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading rather than multiprocessing because threads share a common memory area. They don't allocate a separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

## II. Thread in Java

A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate paths of execution. That's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. All threads of a process share the common memory. The process of executing multiple threads simultaneously is known as multithreading.

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

### **Here are some key points about threads in Java:**

**Multithreading:** Java provides built-in support for multithreading, allowing you to create and manage multiple threads within a single process.

**Concurrency:** Threads allow parts of a program to run concurrently, executing tasks independently and potentially improving the overall performance of the application.

**Parallelism:** Multithreading can lead to parallel execution, where multiple threads are executed simultaneously on multiple processor cores, further enhancing performance.

**Thread States:** Threads can be in various states such as "New," "Runnable," "Blocked," "Waiting," "Timed Waiting," and "Terminated." These states define what a thread is doing at a particular point in time.

**Lifecycle:** A thread goes through different stages in its lifecycle, including creation, running, waiting, and termination. You can control thread behavior using methods like start(), sleep(), wait(), and notify().

**Synchronization:** When multiple threads access shared resources concurrently, synchronization mechanisms like locks, semaphores, and monitors are used to ensure data consistency and prevent race conditions.

**Daemon Threads:** Daemon threads are background threads that are terminated when all non-daemon threads have finished executing. They're used for tasks like garbage collection or logging.

**Thread Safety:** Writing multithreaded code requires careful consideration to ensure that shared data is accessed safely by multiple threads. Techniques like synchronization and thread-safe data structures are used to achieve thread safety.

**Java Concurrency Utilities:** Java provides a rich set of concurrency utilities, including the Executor framework for managing thread pools, the ConcurrentHashMap for thread-safe map operations, and the CountDownLatch for synchronizing threads.

Threads in Java are a fundamental part of building responsive and efficient applications, especially in scenarios where tasks can be executed in parallel or concurrently. However, multithreading also introduces challenges like race conditions and deadlocks, so proper design and synchronization are crucial for writing reliable and efficient multithreaded code.

## **Advantages of Java Multithreading**

- 1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- 2) You can perform many operations together so it saves time.
- 3) Threads are independent so it doesn't affect other threads if exceptions occur in a single thread.

## **Creating a thread in Java**

There are two ways to create a thread in Java:

- 1) By extending Thread class.
- 2) By implementing a Runnable interface.

### **1. Using Thread Class**

#### **Constructors and methods of the Thread class:**

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String) name)

#### **Here's a list of some important methods provided by the Thread class in Java:**

**start():** Initiates the execution of the thread by invoking its run() method.

**run():** Contains the code that the thread will execute. Override this method to define the thread's behavior.

**sleep(long millis):** Pauses the execution of the current thread for the specified number of milliseconds.

**yield():** Suggests that the thread scheduler should consider moving to another thread. The effect is not guaranteed and depends on the scheduler's implementation.

**join():** Waits for the thread to complete its execution. The calling thread will block until the joined thread finishes.

**join(long millis):** Similar to join(), but with a timeout. The calling thread will wait for the specified time before continuing, regardless of whether the joined thread completes.

**interrupt():** Interrupts the thread, causing it to throw an InterruptedException if it's currently waiting or sleeping. It's a way to request a thread to stop gracefully.

**isInterrupted():** Checks if the thread has been interrupted.

**currentThread():** Returns a reference to the currently executing thread.

**getName():** Returns the name of the thread.

**setName(String name):** Sets the name of the thread.

**getId():** Returns the unique identifier of the thread.

**getPriority():** Returns the priority of the thread (a value between Thread.MIN\_PRIORITY and Thread.MAX\_PRIORITY).

**setPriority(int priority):** Sets the priority of the thread.

**isAlive():** Checks if the thread is still running (i.e., not terminated).

**getState():** Returns the current state of the thread.

**isDaemon():** Checks if the thread is a daemon thread.

**setDaemon(boolean on):** Sets the thread to be a daemon or non-daemon thread.

**suspend():** Deprecated. Used to suspend the execution of the thread.

**resume():** Deprecated. Used to resume the execution of a suspended thread.

**stop():** Deprecated. Used to forcefully terminate a thread's execution, which can leave resources in an inconsistent state.

These are just some of the important methods provided by the Thread class. When working with threads, it's important to understand how to manage their execution, handle synchronization, and ensure proper termination for a smooth and reliable multithreading experience.

## Defining, Instantiating, and Starting Threads

### Defining a Thread:

Defining a thread involves creating a class that either extends the Thread class or implements the Runnable interface. The thread's behavior is defined by overriding the run() method within this class. This method contains the code that will be executed when the thread runs.

### Instantiating a Thread:

To instantiate a thread, you create an object of the thread class you defined earlier. This object represents the thread and encapsulates its behavior. If implementing Runnable, the thread object can be constructed by passing an instance of the class that implements Runnable to the Thread constructor.

### Starting a Thread:

Starting a thread involves invoking the start() method on the thread object. This method initiates the execution of the run() method in a separate thread. The thread's execution happens concurrently with the main program or other threads. The order of execution between threads is not guaranteed, and they can run in parallel if the system has multiple processors or cores.

In summary, defining a thread involves creating a class that defines its behavior, instantiation involves creating an instance of this class, and starting the thread causes the run() method to be executed concurrently in a separate thread. Threads enable concurrent execution, improving program efficiency and responsiveness.

### Define a Thread Class:

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface. Implementing Runnable is generally recommended as it separates the thread's behavior from the thread itself.
- You can extend the Thread class

### Implementing Thread step by step:

#### Step 1: Create a Thread Class by extending Thread class

Create a new class that extends the Thread class. Override the run() method in your class to define the behavior of the thread.

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        // Code to be executed in the thread  
    }  
}
```

```

        System.out.println("Thread is running.");
    }
}

```

## Step 2: Instantiating a Thread

In the main program or in another class, instantiate a custom thread class and call the `start()` method to begin execution.

```

public class ThreadExample {
    public static void main(String[] args) {
        // Step 2: Instantiate and start the thread
        MyThread myThread = new MyThread();
        myThread.start();

        // The main thread continues running concurrently with the new thread
        System.out.println("Main thread is running.");
    }
}

```

## Step 3: Starting a Thread

### Start the Thread

Call the `start()` method on the Thread object to start the thread. The `run()` method defined in your Runnable instance will be executed concurrently.

```
thread.start();
```

**Note:** The `run()` method of the thread will be executed concurrently when you call `start()`. You can have multiple threads running concurrently by creating and starting multiple instances.

**Here is a complete example that shows how to make a thread by extending Thread class:**

```

package threadexamples;
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running.");
    }
}

```

```

public class ThreadExample {

```

```

public static void main(String[] args) {

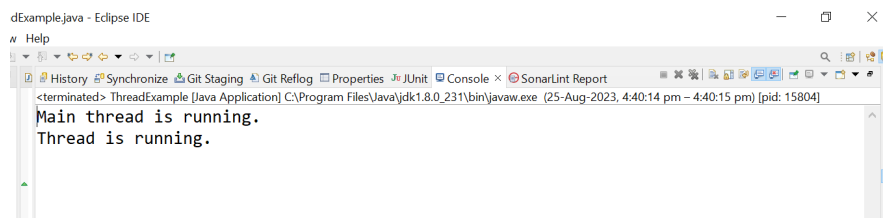
    // Step 2: Instantiate and start the thread
    MyThread myThread = new MyThread();
    myThread.start();

    // The main thread continues running concurrently with the new thread
    System.out.println("Main thread is running.");

}
}

```

## Output:



## 2. Using Runnable Interface

In Java, the Runnable interface is an essential part of multithreading and concurrency. It defines a single abstract method called `run()`, which encapsulates the code to be executed by a thread. By implementing the Runnable interface, you can create threads that execute the `run()` method concurrently.

### Benefits of Runnable:

- Implementing Runnable provides a way to separate the code that needs to run concurrently from the threading logic.
- It's a more flexible and preferred approach to creating threads compared to extending the Thread class directly.
- You can reuse the same Runnable instance for multiple threads if needed.

Using the Runnable interface is a fundamental technique for implementing multithreading in Java, and it promotes better code organization and separation of concerns. It allows you to write thread-safe code by encapsulating the task you want to execute concurrently within the `run()` method.

**Here's a brief overview of how the Runnable interface works:**

**Step 1. Defining a MyRunnable class by implementing Runnable Interface:**



```

public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running.");
    }
}

```

**Step 2: Create an instance of your thread class and create a Thread object.**

```

MyRunnable myRunnable = new MyRunnable();

```

**Step 3: Creating a Thread:**

To execute the run() method concurrently, you need to wrap your Runnable object in a Thread object:

```

Thread thread = new Thread(myRunnable);

```

**Step 4: Starting the Thread:**

After creating the Thread object, you call the start() method to initiate the execution of the run() method in a new thread:

```

thread.start();

```

**Here is a complete example that shows how to make a thread using the Runnable interface:**

```

package threadexamples;
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed in the thread
        System.out.println("Thread is running.");
    }
}

public class ThreadExample {
    public static void main(String[] args) {

        // Step 2: Instantiate the Runnable
        MyRunnable myRunnable = new MyRunnable();

        // Step 3: Create a Thread object
        Thread thread = new Thread(myRunnable);
    }
}

```

```
// Step 4: Start the Thread
thread.start();
// The main thread continues running concurrently with the new thread
System.out.println("Main thread is running.");
}
}
```

## Output:



## Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, running, and dead.

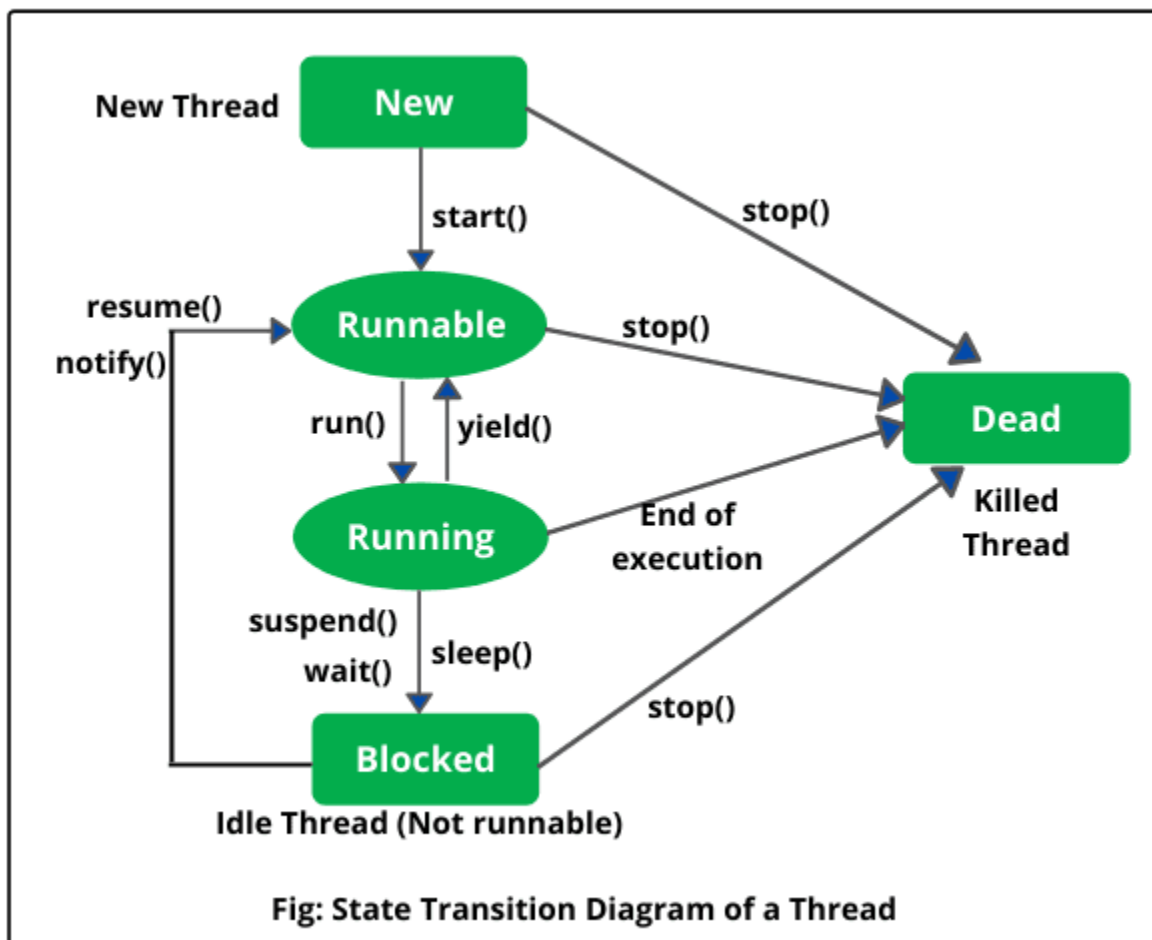
**In Java, a thread always exists in any one of the following states. these states are:**

- **New:** A thread is in the "New" state when it's just created using the new Thread() constructor. At this point, the thread has been initialized but has not yet started executing.
- **Runnable:** Once the start() method is called on the thread object, it enters the "Runnable" state. In this state, the thread is ready to run, but it might not actually be running at the moment due to the scheduling by the operating system.
- **Running:** When the thread scheduler selects the thread for execution, it enters the "Running" state. The actual code of the thread is executed in this state.
- **Blocked/Waiting:** Threads can transition to the "Blocked" or "Waiting" state when they are waiting for a certain condition to be fulfilled, such as waiting for I/O operations, synchronization locks, or explicit calls to wait for signals from other threads.
- **Timed Waiting:** Threads can also enter the "Timed Waiting" state when they are waiting for a specific duration of time. This can happen when they call methods like sleep() or wait() with a timeout.

- **Terminated:** A thread enters the "Terminated" state when it completes its execution or if an unhandled exception occurs during its execution. Once terminated, a thread cannot be restarted.

These states form a cycle, where a thread starts in the "New" state, progresses through the "Runnable" and "Running" states, might enter "Blocked" or "Waiting" states, then returns to "Runnable" or "Running," and eventually reaches the "Terminated" state.

The following diagram shows the complete life cycle of a thread:



### Practical demonstration of Thread states

```

package threadexamples;
class MyThread1 extends Thread {
@Override
public void run() {
System.out.println("Thread is running.");
}
}
  
```

```

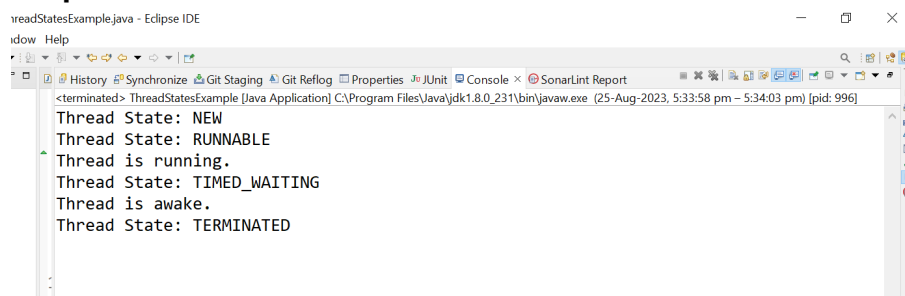
synchronized (ThreadStatesExample.class) {
try {
Thread.sleep(2000);
ThreadStatesExample.class.wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
System.out.println("Thread is awake.");
}
}

public class ThreadStatesExample {
    public static void main(String[] args) throws InterruptedException {

        Thread thread = new MyThread1();
        System.out.println("Thread State: " + thread.getState()); // NEW
        thread.start();
        System.out.println("Thread State: " + thread.getState()); // RUNNABLE
        Thread.sleep(1000);
        System.out.println("Thread State: " + thread.getState()); // TIMED_WAITING
        synchronized (ThreadStatesExample.class) {
            ThreadStatesExample.class.notify();
        }
        Thread.sleep(1000);
        System.out.println("Thread State: " + thread.getState()); // TERMINATED
    }
}

```

## Output:



```

ThreadStatesExample.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> ThreadStatesExample [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (25-Aug-2023, 5:33:58 pm - 5:34:03 pm) [pid: 996]
Thread State: NEW
Thread State: RUNNABLE
Thread is running.
Thread State: TIMED_WAITING
Thread is awake.
Thread State: TERMINATED

```

In this example, we define a separate class `MyThread1` that extends the `Thread` class and overrides the `run()` method. The main thread's behavior remains the same as in the previous example, observing the thread states and transitions:

The thread starts in the "NEW" state after instantiation.

When started, the thread becomes "RUNNABLE".

After 1 second of sleep, the thread enters "TIMED\_WAITING" as it's sleeping.

After 2 seconds, the thread enters a waiting state by calling `wait()` and releases the monitor.

The main thread calls `notify()` on the monitor, awakening the waiting thread.

The thread completes its execution and enters the "TERMINATED" state.

Using a separate class for the thread's behavior provides a clearer view of the code and is particularly helpful when dealing with more complex thread logic.

### III. Thread Scheduler in Java

Thread scheduler is a component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

**Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

#### Thread Priority:

In Java, the `setPriority(int priority)` method is used to set the priority of a thread. This method allows you to assign a priority value to a thread, where priority is an integer representing the desired priority level. Priorities range from 1 (the lowest) to 10 (the highest), and there are three predefined constants available for thread priorities:

`Thread.MIN_PRIORITY` (1): Represents the minimum priority.

`Thread.NORM_PRIORITY` (5): Represents the default (normal) priority.

`Thread.MAX_PRIORITY` (10): Represents the maximum priority.

Here's the syntax for using the `setPriority()` method:

```
public final void setPriority(int priority)
```

**priority:** An integer value between 1 and 10, inclusive, specifying the thread's priority level.

**Note:** We will provide examples exclusively during this session.

**Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **the arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

### Thread Scheduler Algorithms

On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

### **1. First Come First Serve Scheduling:**

In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.



### First Come First Serve Scheduling

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

## 2. Time-slicing scheduling:

Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

**Note:** In multithreading, "infinite blocking," also known as "starvation," refers to a situation where a thread in a concurrent program is unable to make progress because it is consistently denied access to a resource or a critical section of code. In other words, the thread becomes "blocked" indefinitely and can't proceed with its execution.

Starvation typically occurs when a scheduling or resource allocation policy is unfair or biased towards certain threads, causing them to monopolize resources or execution time. As a result, other threads, despite being ready to run, are continually postponed or delayed.

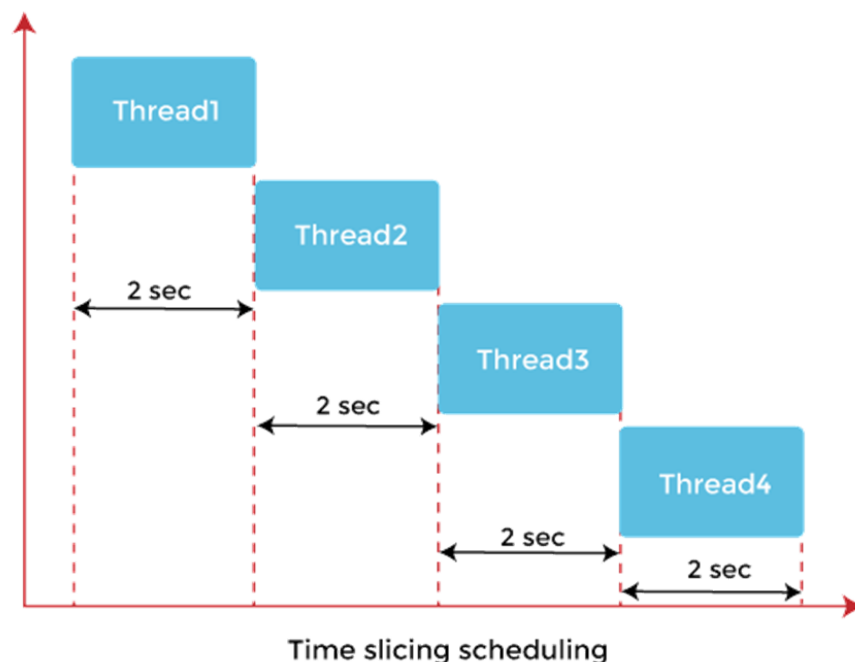
**Common scenarios that can lead to starvation include:**

**Priority-Based Scheduling:** If a higher-priority thread continually preempts lower-priority threads, the lower-priority threads may be starved of CPU time and unable to execute.

**Resource Contention:** In situations where multiple threads are competing for limited resources (e.g., locks, semaphores), if one or more threads consistently acquire and hold the resources for extended periods, other threads may be starved and unable to access those resources.

**Unfair Locking:** If a locking mechanism doesn't employ a fair policy, it may allow some threads to repeatedly acquire a lock while others are perpetually denied access, resulting in those threads being starved.

To address starvation issues, it's important to design multithreaded applications and scheduling policies carefully. Fair scheduling policies, such as round-robin or priority-based scheduling with aging, can help ensure that all threads get a fair opportunity to execute. Additionally, proper resource management, efficient locking strategies, and avoiding excessive thread priority disparities are essential to mitigate the risk of infinite blocking or starvation in multithreaded systems.



In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

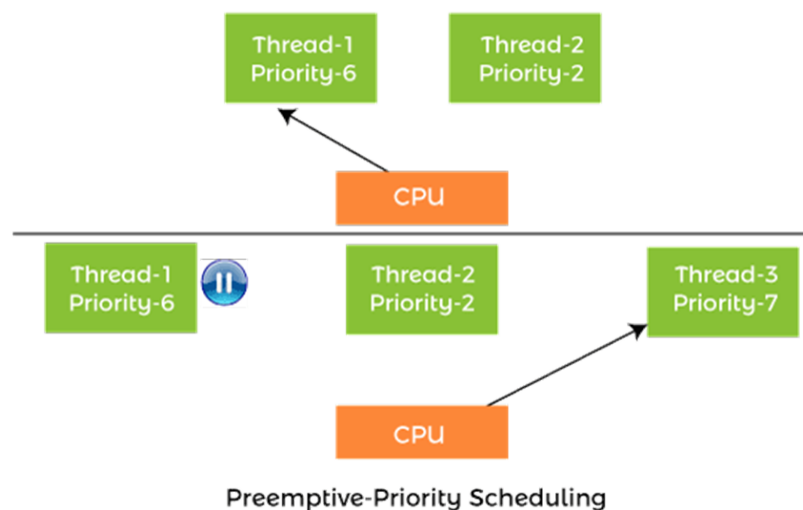


**Note:** Preemptive scheduling allows the operating system to forcibly interrupt a running thread in favor of a higher-priority one, ensuring fairness and responsiveness in multitasking environments.

Non-preemptive scheduling relies on threads voluntarily releasing the CPU and can be simpler but may lead to potential issues if threads do not cooperate effectively. Most modern operating systems employ preemptive scheduling to provide efficient and predictable multitasking.

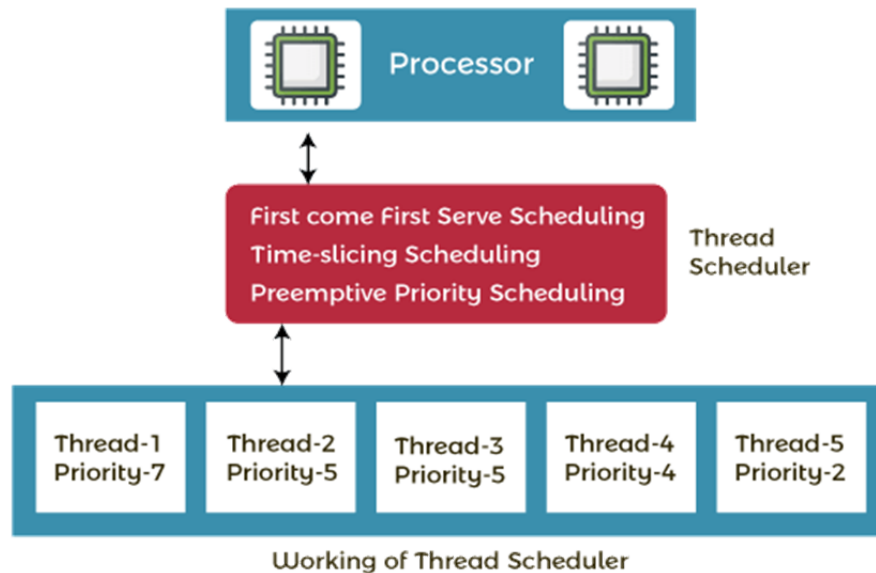
### 3. Preemptive-Priority Scheduling:

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

## Working of the Java Thread Scheduler



Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is preempted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) have the same priorities and arrival time, the scheduling will be decided on the basis of the FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

**Let's explore an example of a thread with priority.**

```
package multithreading;
class MyThread extends Thread {
    public MyThread(String name, int priority) {
        super(name);
        setPriority(priority);
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + ": Count " + i);
            try {
```

### Output:

In this example we create a `MyThread` class that extends `Thread`. This class takes a name and a priority as constructor parameters and sets the thread's priority using the `setPriority()` method. In the `run()` method of `MyThread`, the thread counts from 1 to 5, printing its name and the count for each iteration. It also simulates some work with a short sleep.

In the main method, we create three threads with different priorities: one with maximum priority, one with normal priority (the default), and one with minimum priority. We start all three threads using the start() method.

The thread with maximum priority (Thread.MAX\_PRIORITY) is more likely to get CPU time and finish its work earlier than the others. However, thread priority doesn't guarantee strict execution order, and the actual behavior may vary depending on the operating system and the thread scheduler.

### **Let's explore an example of Naming Thread using Constructor and join() method:**

```
package multithreading;
public class ThreadDemo extends Thread{
    //Using constructor to name the thread
    public ThreadDemo(String name)
    {
        super(name);
    }
    //here we are running loop from 1 to 5 to display from 1 to 5
    public void run()
    {
        for(int i=0;i<=5;i++)
        {
            try {
                Thread.sleep(400); //invoking sleep() method for context switching between two thread
            }catch (InterruptedException e) {
                System.out.println(e);
            }
            //Displaying values of "i" with the name of thread
            System.out.println("Thread: "+Thread.currentThread().getName()+" :"+ i);
        }
    }

    public static void main(String[] args) {
        //Creating 3 threads with name t1,t2 and t3
        ThreadDemo t1=new ThreadDemo("t1");
        ThreadDemo t2=new ThreadDemo("t2");
        ThreadDemo t3=new ThreadDemo("t3");

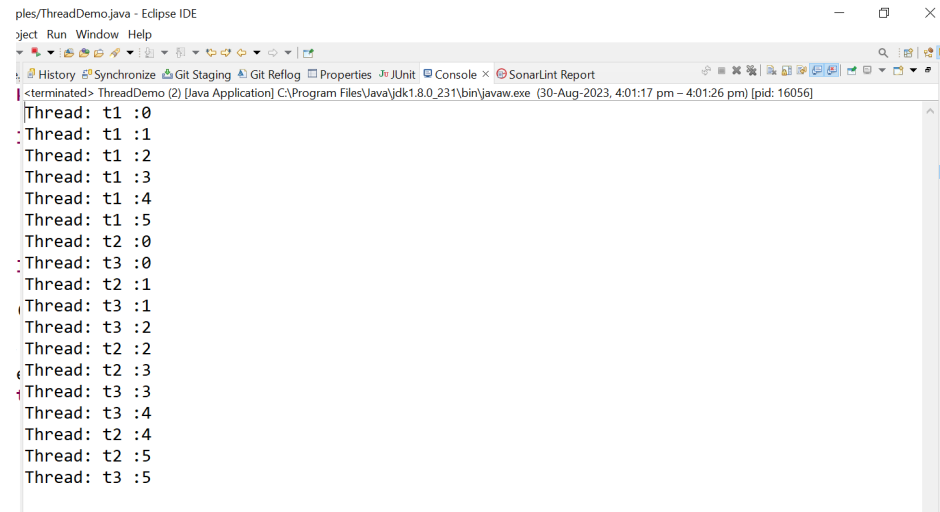
        t1.start();
        try {
            t1.join();// Calling join() method by t1 so that t2 and t3 will wait for this
            thread "t1" to die
        }catch (Exception e) {
        }
    }
}
```

```

        //After t1 then we are starting t2 and t3
        t2.start();
        t3.start();
    }
}

```

## Output:



```

ThreadDemo.java - Eclipse IDE
ject Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
terminated> ThreadDemo (2) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (30-Aug-2023, 4:01:17 pm - 4:01:26 pm) [pid: 16056]
Thread: t1 :0
Thread: t1 :1
Thread: t1 :2
Thread: t1 :3
Thread: t1 :4
Thread: t1 :5
Thread: t2 :0
Thread: t3 :0
Thread: t2 :1
Thread: t3 :1
Thread: t3 :2
Thread: t2 :2
Thread: t2 :3
Thread: t3 :3
Thread: t3 :4
Thread: t2 :4
Thread: t2 :5
Thread: t3 :5

```

In this example we create a ThreadDemo class that extends Thread. This class takes a name as constructor parameters and sets the name as a Thread name. In the run() method of ThreadDemo, the thread prints from 1 to 5, printing its name for each iteration. It also simulates some work with a short sleep.

In the main method, we instantiate three threads with distinct names. Subsequently, we utilize the join() method with the t1 object, causing the current thread to pause and wait for t1 to complete its execution. This means the calling thread remains in a blocked state until the joined thread, t1, finishes its task. Afterward, we initiate both t2 and t3 using the start() method, enabling them to work through context switching (meaning once t2 will execute then t3 and it will continue till the last execution).

## Knowledge check...

***Now that we've covered Thread, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.***

**1. What is the name of the method used to start a thread execution?**

- A. run();
- B. init();
- C. start();
- D. resume();

**2.Which method must be defined by a class implementing the java.lang Runnable interface?**

- A. public void run()
- B. void run()
- C. void run(int priority)
- D. public void start()

***At the end of the quiz,the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.***

#### **IV. Synchronization**

Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.

When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.

To implement the synchronous behavior Java has a synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread comes out of the synchronized block.

When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In Java we have the solution for this, put the calls to the methods (which need to be synchronized) defined by this class inside a synchronized block in the following manner.

#### **Why use Synchronization?**

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problems.

#### **Types of Synchronization**

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

**Note:** Java uses Thread Synchronization.

## Thread Synchronization

There are two types of thread synchronization: Mutual exclusive and inter-thread communication.

1. Mutual Exclusive
  1. Synchronized method.
  2. Synchronized block.
  3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

```
wait()  
notify()  
notifyAll()
```

## Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

## Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

## Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

## Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

**Let's see the example:**

```
package synchronization;
```

```
class Factorial
{
//here we have created synchronized method fact() which will calculate factorial
synchronized void fact(int n)
    {
        int f=1;
        for(int i=1;i<=n;i++)
        {
            f=f*i;
            System.out.println("Thread: "+Thread.currentThread().getName()+" : "+ f);
            try {
                Thread.sleep(400);
            }catch (Exception e)
            {
                System.out.println(e);
            }
        }
        System.out.println("Factorial of "+ n+" is:"+f);
    }
}
```



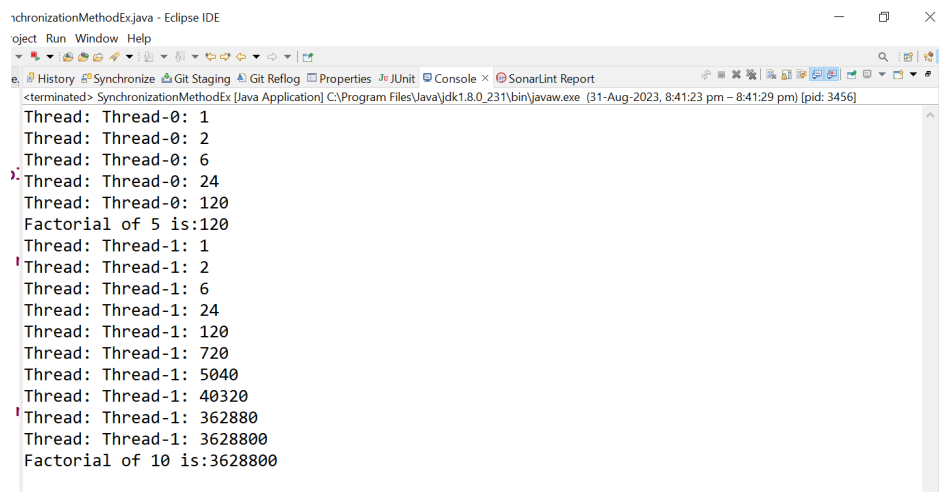
```

}
public class SynchronizationMethodEx {
    public static void main(String[] args) {
//created object of Factorial class and lock is associated with this object
        final Factorial fact=new Factorial();

//here we are creating anonymous objects with different value to calculate factorial
new Thread()
    {
        public void run()
        {
            fact.fact(5);
        }
    }.start();
new Thread() {
    public void run()
    {
        fact.fact(10);
    }
}.start();
    }
}

```

## Output



The screenshot shows the Eclipse IDE interface with the console window open. The console output displays the execution of the SynchronizationMethodEx program. It shows two threads, Thread-0 and Thread-1, each calculating the factorial of a number. Thread-0 calculates the factorial of 5, resulting in 120. Thread-1 calculates the factorial of 10, resulting in 3628800. The output is as follows:

```

<terminated> SynchronizationMethodEx [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (31-Aug-2023, 8:41:23 pm - 8:41:29 pm) [pid: 3456]
Thread: Thread-0: 1
Thread: Thread-0: 2
Thread: Thread-0: 6
Thread: Thread-0: 24
Thread: Thread-0: 120
Factorial of 5 is:120
Thread: Thread-1: 1
Thread: Thread-1: 2
Thread: Thread-1: 6
Thread: Thread-1: 24
Thread: Thread-1: 120
Thread: Thread-1: 720
Thread: Thread-1: 5040
Thread: Thread-1: 40320
Thread: Thread-1: 362880
Thread: Thread-1: 3628800
Factorial of 10 is:3628800

```

In this example, we have created two anonymous thread classes in the main class and both have a method run() that is void. Both are printing factorial values of different numbers and method start() is invoked.

## Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

If a block is declared as synchronized then the code which is written inside a method is only executed instead of the whole code. It is used when sequential access to code is required.

If we put all the codes of the method in the synchronized block, it will work the same as the synchronized method.

### Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of the synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keywords.
- Java synchronized block is more efficient than Java synchronized method.

### Let's see the example:

```
package synchronization;
class Table
{
//here we have created synchronized method printTable() which will print the product of
n
void printTable(int n){
synchronized(this){ //This is a synchronized block
for(int i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}
}
//end of the synchronized block
}

//creating MyThread1 which is extending Thread
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
```

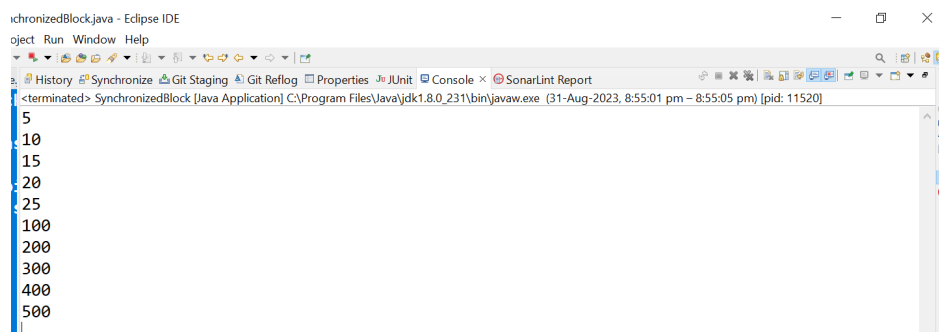
```

t.printTable(5);
}
}
//creating MyThread2 which is extending Thread
class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(100);
    }
}

public class SynchronizedBlock{
    public static void main(String args[]){
        //Created object of table and lock is associated with it
        Table obj = new Table();
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

## Output:



```

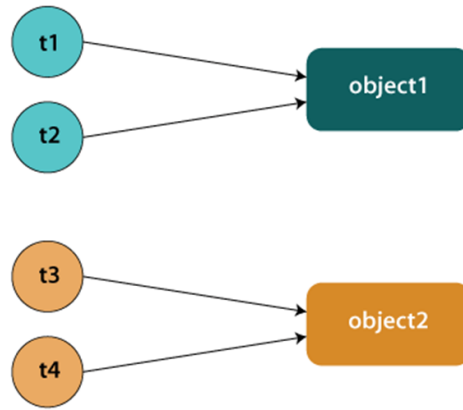
SynchronizedBlock [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (31-Aug-2023, 8:55:01 pm - 8:55:05 pm) [pid: 11520]
5
10
15
20
25
100
200
300
400
500

```

In this example, class thread1 extends the class thread and has a method run that is void. The class thread2 extends the thread and has a method run that is void. Both are printing tables of different numbers. In the main class, the object of threads 1 and 2 are created and the method start is invoked.

## Static Synchronization

If you make any static method as synchronized, It means that lock is applied to the class instead of an object and only one thread will access that class at a time.



### Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Product) named p1 and p2. In the case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refer to a common object that has a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

#### Example:

```
package synchronization;
class Product {
//creating printProduct() method which will print the product of n
synchronized static void printProduct(int n)
{
    for(int i=1;i<=n;i++)
    {
        System.out.println("Thread: "+Thread.currentThread().getName()+" : "+ n*i);
        try {
            Thread.sleep(400);
        }catch (Exception e) {
            System.out.println(e);
        }
    }
}
}
}
public class StaticSynchronizationDemo {
    public static void main(String[] args) {
```

//creating 4 threads anonymous objects with name t1,t2,t3 and t4

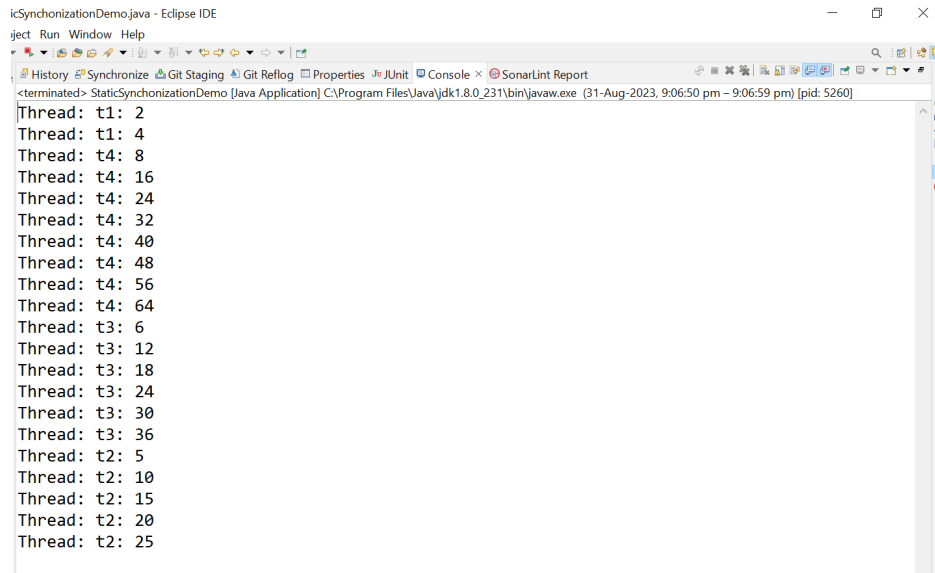
```
new Thread("t1")
{
    public void run()
    {
        Product.printProduct(2);
    }
}.start();

new Thread("t2")
{
    public void run()
    {
        Product.printProduct(5);
    }
}.start();

new Thread("t3")
{
    public void run()
    {
        Product.printProduct(6);
    }
}.start();

new Thread("t4")
{
    public void run()
    {
        Product.printProduct(8);
    }
}.start();
}
```

**Output:**



```
icSynchronizationDemo.java - Eclipse IDE
ject Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> StaticSynchronizationDemo [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (31-Aug-2023, 9:06:50 pm - 9:06:59 pm) [pid: 5260]
Thread: t1: 2
Thread: t1: 4
Thread: t4: 8
Thread: t4: 16
Thread: t4: 24
Thread: t4: 32
Thread: t4: 40
Thread: t4: 48
Thread: t4: 56
Thread: t4: 64
Thread: t3: 6
Thread: t3: 12
Thread: t3: 18
Thread: t3: 24
Thread: t3: 30
Thread: t3: 36
Thread: t2: 5
Thread: t2: 10
Thread: t2: 15
Thread: t2: 20
Thread: t2: 25
```

In this example, we have created four anonymous thread classes in the main class and both have a method `run()` that is void. tables of different numbers. In the main class, the object of threads 1 and 2 are created and the method `start` is invoked. In this example, Lock is associated with class instead of object. So at a time only one thread can access the `printProduct()` method.

**Question:** Have you ever experienced a situation where a lack of synchronization caused a problem in your daily life (e.g., double-booking appointments or conflicting schedules)? Share the experience and discuss how synchronization could have prevented it.

## V. Inter-thread Communication(Cooperation)

When a program is executed, it becomes a process, which can be further divided into threads. Threads are independent paths of execution within the same process, and they allow a program to perform multiple tasks simultaneously, which can improve performance and efficiency.

Inter-Thread Communication (Cooperation) is a mechanism that allows threads to exchange information or coordinate their execution. It enables threads to work together to solve a common problem or to share resources. ITC can involve synchronization mechanisms such as mutexes or semaphores, which ensure that critical sections of code are executed by only one thread at a time to avoid concurrency issues.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- `wait()`
- `notify()`

- `notifyAll()`

### 1) `wait()` method

Let's say we are currently running Thread1 and we want to run Thread2. Since inter thread communication can be done in a synchronized block only one thread can run at a time. So to run Thread2 we must "pause" Thread1. The `wait()` function helps us achieve this exact thing.

The `wait()` method aids inter thread communication by releasing the lock on the current or calling thread (Thread1 in the above example) and instructing it to sleep until another thread (Thread2 in the above example) enters the monitor and calls `notify()` or `notifyAll()`, or until a certain period of time has passed.

The current thread must own this object's monitor, hence the `wait()` function must be used from the synchronized method only; otherwise, an error will be thrown.

#### Syntax:

Method	Description
<code>public final void wait() throws InterruptedException</code>	It waits until the object is notified.
<code>public final void wait(long timeout) throws InterruptedException</code>	It waits for the specified amount of time.

### 2) `notify()` method

Now let's say after we completed our work in Thread2 and we want to return to Thread1. To do so, we need to inform Thread1 that it can now use the object to run itself. We can use the `notify()` method to do so.

NOTE: Why is Thread1 waiting for the object?

- Because we used the `wait()` function in Thread1 and since only one thread can run at a time Thread1 is waiting for the object to run.

The `notify` method() aids inter-thread communication by allowing a single thread that is waiting for the object's monitor to become a runnable thread by waking it up.

Note: The decision of which waiting thread to awaken is arbitrary and is determined by the implementation.

#### Syntax:

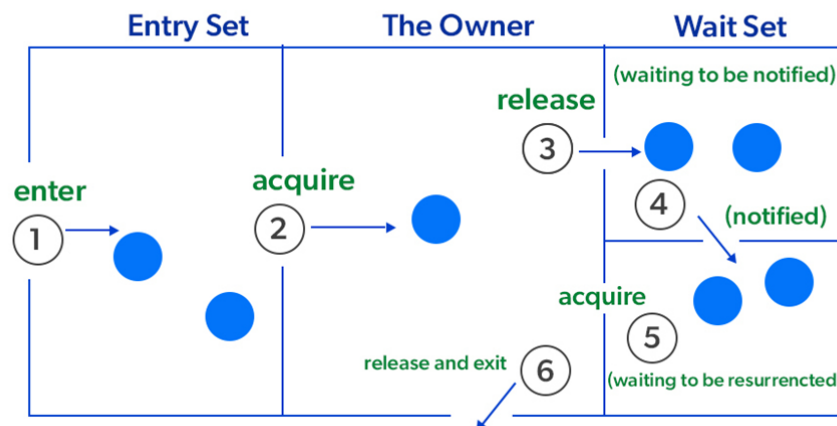
```
public final void notify()
```

### 3) `notifyAll()` method

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

**public final void** notifyAll()



1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, the thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

**Let's explore with an Example:**

```
package synchronization;
//creating Accounts class with data member balance 20000
class Accounts
{
    int balance=20000;

    //created synchronized withdraw () method which will withdraw the amount
    synchronized void withdraw(int amount)
    {
        System.out.println("going to withdraw "+amount+" rs.");
        if(balance<amount)
        {
            System.out.println("insufficient balance!!waiting for deposit....");
            try {
```



```

        wait(); //waiting thread to notify
    }catch (Exception e) {
        // TODO: handle exception
    }
    this.balance-=amount;
    System.out.println("withdraw done");
}
}

```

//created synchronized deposit() method which will withdraw the amount

```

synchronized void deposit(int amount)
{
    System.out.println(amount+" going to deposit");
    this.balance+=amount;
    System.out.println("deposit done");
    notify(); //notifying withdraw() method which is waiting for
}
}

```

```

public class InterThreadCommunication {
    public static void main(String[] args) {
        Accounts acc=new Accounts();//lock is associated with acc object.

```

//created two anonymous thread object to call withdraw() and deposit()

method

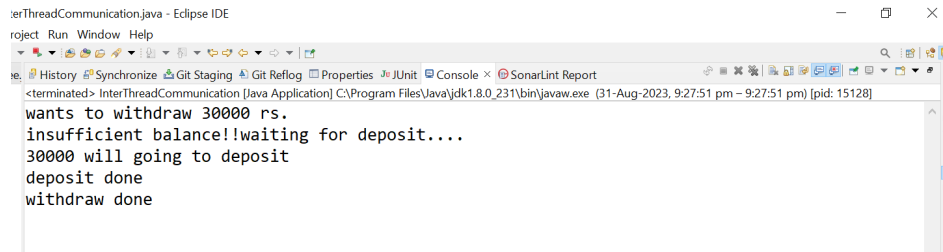
```

        new Thread()
        {
            public void run()
            {
                acc.withdraw(30000);
            }
        }.start();

        new Thread()
        {
            public void run()
            {
                acc.deposit(30000);
            }
        }.start();
    }
}

```

**Output:**

The screenshot shows the Eclipse IDE's console window. The title bar indicates the file is 'erThreadCommunication.java'. The console output shows a sequence of messages: 'wants to withdraw 30000 rs.', 'insufficient balance!!waiting for deposit...', '30000 will going to deposit', 'deposit done', and 'withdraw done'. The console also shows a status bar at the bottom indicating the application is terminated and the PID is 15128.

```
erThreadCommunication.java - Eclipse IDE
roject Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> InterThreadCommunication [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (31-Aug-2023, 9:27:51 pm - 9:27:51 pm) [pid: 15128]
wants to withdraw 30000 rs.
insufficient balance!!waiting for deposit...
30000 will going to deposit
deposit done
withdraw done
```

In this code, we've created an Accounts class representing a bank account with a shared balance field. The class has two synchronized methods, withdraw and deposit, ensuring only one thread can access them at a time to prevent data inconsistencies. When a thread calls withdraw, it checks if there's enough balance for the requested withdrawal. If not, it enters a waiting state using wait() until another thread deposits money and notifies it. When a thread calls deposit, it adds the specified amount to the balance and uses notify() to wake up any waiting threads.

In the main method, we create an Accounts object and start two threads concurrently. One thread simulates a withdrawal, and the other simulates a deposit.

**Note:** This code demonstrates safe concurrent access to a shared resource using synchronization and inter-thread communication.

## VI. JVM (Overview)

JVM stands for "Java Virtual Machine." It is a fundamental component of the Java programming language and runtime environment. Here's what JVM is and its key functions:

**Execution Environment:** JVM serves as a virtualized execution environment for Java applications. It allows Java programs to run on different hardware platforms without modification. This "write once, run anywhere" capability is one of Java's key features.

**Interpreting and Compiling:** JVM can interpret Java bytecode or compile it into native machine code, depending on the implementation. Most modern JVMs use a combination of both interpretation and Just-In-Time (JIT) compilation to optimize performance.

**Memory Management:** JVM manages memory allocation and garbage collection. It automatically allocates memory for objects and deallocates memory when objects are no longer in use, preventing memory leaks.

**Platform Independence:** Java source code is compiled into bytecode, which is platform-independent. JVM then interprets or compiles this bytecode for the specific platform it's running on, ensuring portability.

**Security:** JVM includes built-in security features to protect against malicious code. It uses classloaders and bytecode verification to ensure that code adheres to Java's security rules.

**Classloading:** JVM loads classes as they are referenced during program execution. It follows a hierarchical class loading mechanism and ensures that classes are loaded only once.

**Execution of Bytecode:** JVM executes Java bytecode instructions, which are generated by compiling Java source code. It manages method calls, exception handling, and other program operations.

**Multi-Threading and Synchronization:** JVM provides support for multi-threading and synchronization, allowing Java programs to run multiple threads concurrently while managing thread safety.

**Performance Optimization:** JVMs use various techniques, such as Just-In-Time compilation and runtime profiling, to optimize the performance of Java applications.

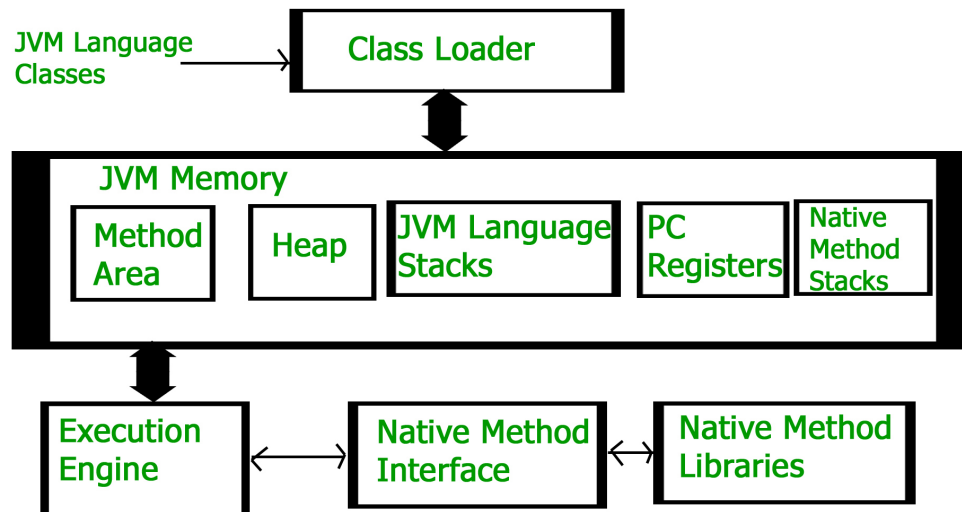
Popular implementations of JVM include Oracle HotSpot, OpenJDK, and IBM J9, among others. Different JVM implementations may have variations in performance and features, but they all adhere to the Java Virtual Machine Specification to ensure compatibility with Java programs.

### **How JVM Works – JVM Architecture:**

JVM(Java Virtual Machine) 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(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

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, generates the corresponding binary data and saves it in the method area. For each “.class” file, JVM stores the following information in the method area.

- The fully qualified name of the loaded class and its immediate parent class.
- Whether the “.class” file is related to Class or Interface or Enum.
- Modifier, Variables and Method information etc.
- 

After loading the “.class” file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package. These Class object can be used by the programmer for getting class level information like the name of the class, parent name, methods and variable information etc.

**Linking:** Performs verification, preparation, and (optionally) resolution.

- **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 a 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.
- **Preparation:** JVM allocates memory for class static variables and initializes the memory to default values.
- **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.

**In general, there are three class loaders :**

- **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++.
- **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$ExtClassLoader` class.
- **System/Application class loader:** It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variables which are mapped to `java.class.path`. It is also implemented in Java by the `sun.misc.Launcher$AppClassLoader` class.

```
// Java code to demonstrate Class Loader subsystem
public class Test {
    public static void main(String[] args)
    {
        // String class is loaded by bootstrap loader, and
```

```

// bootstrap loader is not Java object, hence null
System.out.println(String.class.getClassLoader());

// Test class is loaded by Application loader
System.out.println(Test.class.getClassLoader());
}
}

```

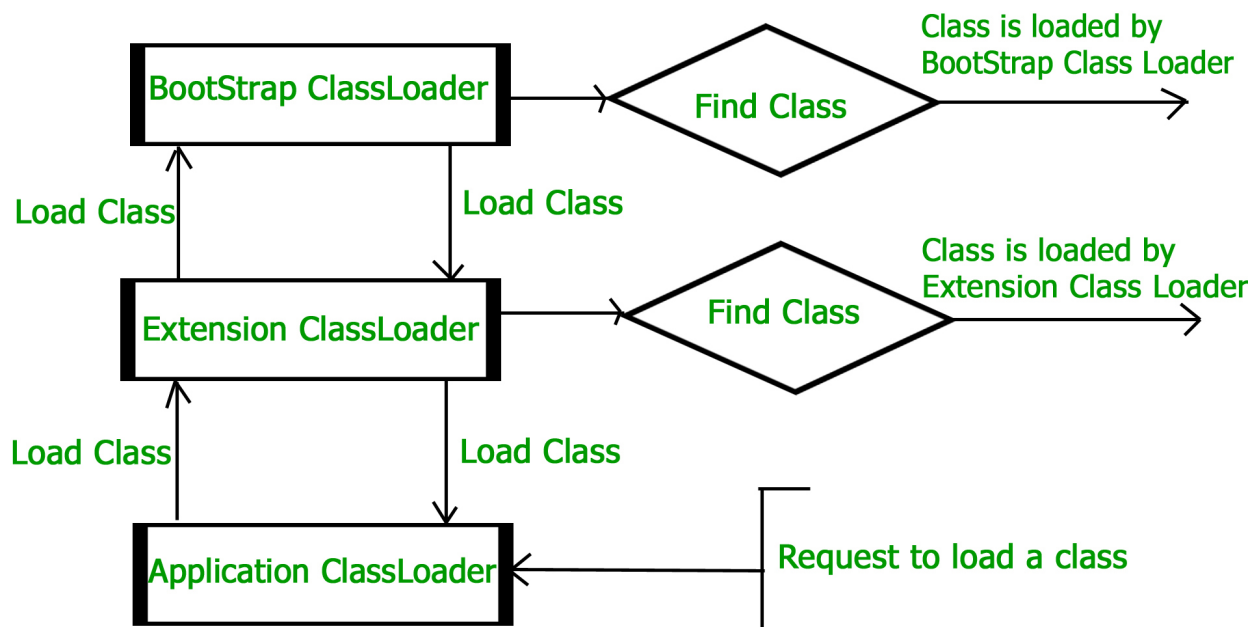
## Output

```

Test.java - Eclipse IDE
Project Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> Test (3) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (07-Sep-2023, 5:31:17 pm - 5:31:21 pm) [pid: 4668]
null
sun.misc.Launcher$AppClassLoader@73d16e93

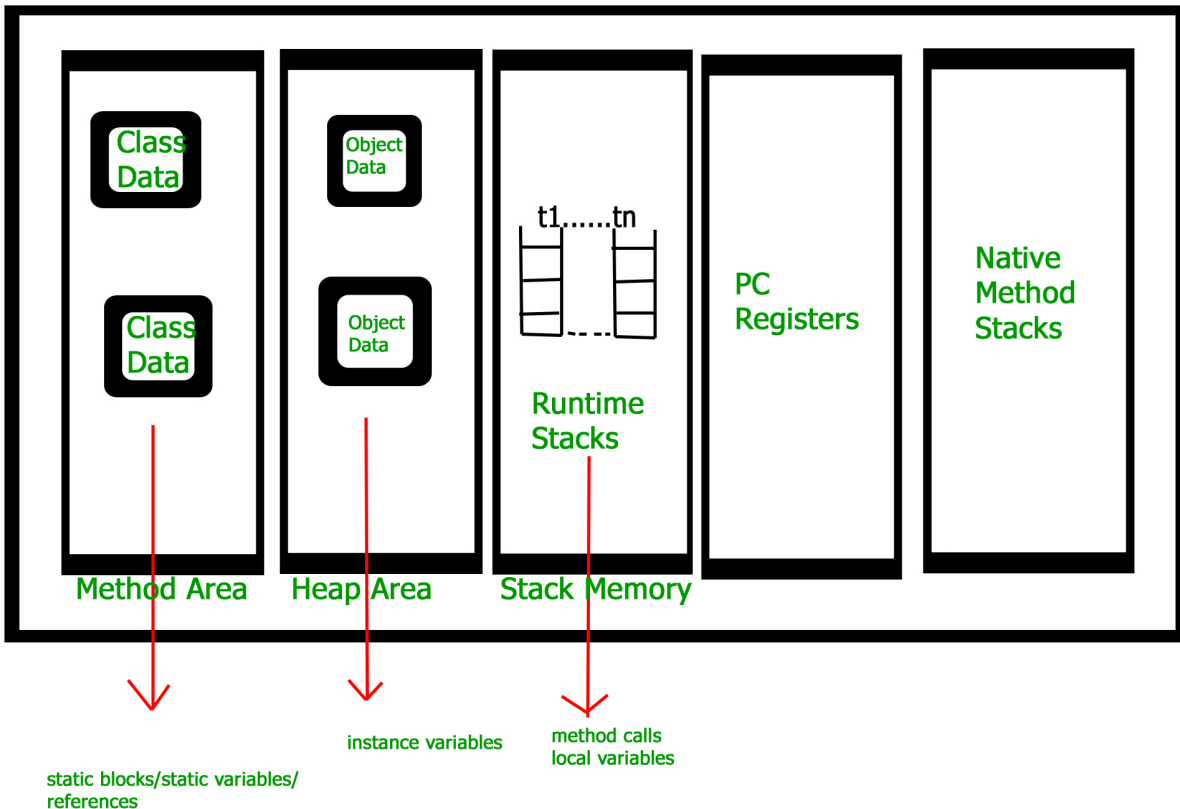
```

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 is found in the boot-strap path, the class is loaded, otherwise the 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 the class, then we get a run-time exception `java.lang.ClassNotFoundException`.



## JVM Memory

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. From java 8, static variables are now stored in Heap area.
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.



## **Execution Engine**

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.

## **Java Native Interface (JNI) :**

It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

## **Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**For further knowledge, do research on ChatGPT**

## **Exercise:**

**In ChatGPT type the following prompt:** What are deadlocks and race conditions in multi-threaded programs? How can they be detected, prevented, and resolved to ensure smooth inter-thread communication?