**Multithreading**

**Introduction to Threads:**

Imagine you're running a web server. Each incoming request from a client can be handled by a separate thread. This allows the server to handle multiple requests simultaneously, improving responsiveness.

**Creating Threads with Thread Class**

Extending the Thread class: You might have a simulation software where each moving object (like a car or a person) is represented by a separate thread, extending the Thread class.

**Example**

We can create a method named show in the classes A and B in the above program. However, since A and B are extending the Thread class, they are treated as separate threads of execution, and calling the show method directly won't run them concurrently. Instead, you'll need to override the run method provided by the Thread class, and then start the threads using the start method, which will internally invoke the run method.

Here's how you can modify the program to include the show method and still run the threads concurrently:

package MultipleThreading;  
  
public class MultipleThreading {  
  
 public static void main(String[] args) {  
 A obj1=new A();  
 B obj2=new B();  
  
// obj1.show();  
// obj2.show();  
  
 obj1.start();  
 obj2.start();  
 }  
}  
  
class A extends Thread {  
 public void show() {  
 for (int i = 1; i <= 10; i++) {  
 System.*out*.println("Hi");  
 }  
 }  
  
 @Override  
 public void run() {  
 show(); // Call the show method from within the run method  
 }  
}  
  
class B extends Thread {  
 public void show() {  
 for (int i = 1; i <= 10; i++) {  
 System.*out*.println("Hello");  
 }  
 }  
  
 @Override  
 public void run() {  
 show(); // Call the show method from within the run method  
 }  
}

**Thread Priority and Sleep**

**Thread Priority:**

Each thread in Java has a priority ranging from 1 to 10, with 1 being the lowest priority and 10 being the highest.

The thread scheduler uses the priority to determine the order in which threads are scheduled to run. Higher priority threads are given preference over lower priority threads.

However, thread priority is only a hint to the scheduler, and the actual behavior may vary depending on the underlying platform and JVM implementation.

You can set the priority of a thread using the **setPriority**() method, and you can get the priority of a thread using the **getPriority**() method.

**Example**:

Thread thread1 = new Thread();

thread1.setPriority(Thread.MIN\_PRIORITY); // Set lowest priority

Thread thread2 = new Thread();

thread2.setPriority(Thread.MAX\_PRIORITY); // Set highest priority

**sleep() Method:**

The **sleep()** method is a static method of the Thread class used to pause the execution of the current thread for a specified amount of time.

It takes a single argument representing the duration of time to sleep in milliseconds. Alternatively, you can use an overloaded version of **sleep()** that takes an additional parameter for nanoseconds.

While a thread is sleeping, it temporarily gives up its execution time, allowing other threads to run. After the specified sleep duration elapses, the thread enters the runnable state and may be scheduled for execution again.

**Example**:

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

// Handle interrupted exception

}

package MultipleThreading;  
  
public class ThreadPriorityAndSleep {  
  
 public static void main(String[] args) {  
 A obj1 = new A();  
 B obj2 = new B();  
  
 // Start both threads  
 obj1.start();  
 obj2.start();  
 }  
}  
  
  
class A extends Thread {  
 public void show() {  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println("Hi");  
 try {  
 // Sleep for 200 milliseconds after printing "Hi"  
 Thread.*sleep*(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 @Override  
 public void run() {  
 // Set thread priority to MIN\_PRIORITY  
 this.setPriority(*MIN\_PRIORITY*);  
 show(); // Call the show method from within the run method  
 }  
}  
  
class B extends Thread {  
 public void show() {  
 for (int i = 1; i <= 5; i++) {  
 System.*out*.println("Hello");  
 try {  
 // Sleep for 300 milliseconds after printing "Hello"  
 Thread.*sleep*(300);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 @Override  
 public void run() {  
 // Set thread priority to MAX\_PRIORITY  
 this.setPriority(*MAX\_PRIORITY*);  
 show(); // Call the show method from within the run method  
 }  
}

**Explanation**:

In this modified example, we have two threads A and B.

Inside each **thread's show()** method, we print a message ("Hi" for thread A and "Hello" for thread B) and then sleep for a specific duration using the **sleep()** method. This simulates some processing time between each message.

Additionally, each **thread's run()** method sets the thread's priority using the **setPriority()** method. Thread A is set to the lowest priority (MIN\_PRIORITY), and thread B is set to the highest priority **(MAX\_PRIORITY).**

When the program is executed, both threads start running concurrently. However, since thread B has a higher priority, it may receive more CPU time compared to thread A. Additionally, the **sleep()** method in each thread introduces a delay between consecutive messages, creating a staggered output.

In a multithreaded environment, the operating system's thread scheduler is responsible for determining which thread to run on the CPU at any given time. The concept of thread priority influences the thread scheduler's decision-making process.

When a thread has a higher priority, it indicates to the thread scheduler that this thread should be given preference over threads with lower priorities when allocating CPU time. In other words, threads with higher priority are more likely to be scheduled to run by the thread scheduler, and they may receive a larger portion of the available CPU time compared to threads with lower priorities.

In the context of the provided program:

Thread A is set to the lowest priority (MIN\_PRIORITY).

Thread B is set to the highest priority (MAX\_PRIORITY).

This means that thread B is signaling to the thread scheduler that it should be given priority over thread A whenever the scheduler needs to make a decision about which thread to run next. As a result, thread B is more likely to be scheduled to run by the thread scheduler, and it may receive more CPU time compared to thread A.

However, it's important to note that thread priority is only a hint to the thread scheduler, and the actual behavior may vary depending on the underlying platform and JVM implementation. Additionally, other factors such as the specific workload, system load, and thread scheduling policies can also influence the distribution of CPU time among threads. Therefore, while higher-priority threads are given preference, it's not guaranteed that they will always receive more CPU time than lower-priority threads.

**Creating Threads with Runnable Interface**

package MultipleThreading;  
  
public class RunableInterfaceEx {  
 public static void main(String[] args) {  
 G obj1= new G();  
 H obj2= new H();  
 Thread t1= new Thread(obj1);  
 Thread t2= new Thread(obj2);  
 t1.start();  
 t2.start();  
  
 }  
}  
  
class G implements Runnable{  
  
public void run(){  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
}  
}  
  
class H implements Runnable{  
 public void run(){  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
 }  
}

Now we convert it in to Lambda Expression as Runnable interface is @FunctionalInterface

Here we use lambda Expression

G obj1= new G();  
H obj2= new H();

Above code can be written as below:

Step1

Runnable obj1= new G();  
Runnable obj2= new H();

Step 2

We have to write override method after the creating object with Runnable Interface with anonymous class. We don’t write the class name directly override the interface method.

Runnable obj1= new Runnable() {  
 @Override  
 public void run() {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
 }  
};

So this class

class G implements Runnable{  
  
public void run(){  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
}  
}

completely replaced by below statement

Runnable obj1= new Runnable() {  
 @Override  
 public void run() {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
 }  
};

And this class

class H implements Runnable{  
 public void run(){  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
 }  
}

completely replaced by below statement

Runnable obj2= new Runnable() {  
 @Override  
 public void run() {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
 }  
};

Step 3

Now this statement

Runnable obj1= new Runnable() {  
 @Override  
 public void run() {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
 }  
};

Can be replaced by

Runnable obj1= () ->  
{  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
};

We will remove method and new Runnable() and replace with ()->{ };

And this statement

Runnable obj2= new Runnable() {  
 @Override  
 public void run() {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
 }  
};

Can be replaced by

Runnable obj2= ()-> {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
};

Please find the whole program

package MultipleThreading;  
  
public class RunableInterfaceEx {  
 public static void main(String[] args) {  
 Runnable obj1= () ->  
 {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hi");  
 }  
 };  
 Runnable obj2= ()-> {  
 for(int i=0;i<=5;i++){  
 System.*out*.println("Hello");  
 }  
 };  
 Thread t1= new Thread(obj1);  
 Thread t2= new Thread(obj2);  
 t1.start();  
 t2.start();  
  
 }  
};

**Race Condition**

A race condition is a phenomenon that occurs in concurrent programming when the outcome of the program depends on the relative timing or interleaving of multiple threads or processes executing concurrently. **In other words, the result of the program becomes unpredictable because the threads' execution order is not guaranteed, leading to unexpected behavior.**

Race conditions typically occur when multiple threads access shared resources or variables concurrently and at least one of the threads performs a write operation. If proper synchronization mechanisms are not in place to coordinate access to these shared resources, the following problems can arise:

**Data Corruption**: When multiple threads concurrently read and write to a shared variable without proper synchronization, the final value of the variable might not be what was expected. This can lead to data corruption and inconsistencies in the program's state.

**Lost Updates**: If multiple threads read, modify, and then write back the value of a shared variable, one thread's updates may be lost due to another thread overwriting the value before it can be saved.

**Deadlocks and Livelocks**: Race conditions can also lead to deadlocks or livelocks if threads become blocked waiting for resources that are held by other threads indefinitely.

**Example:**

Suppose we have a Counter class with a shared count variable that multiple threads will increment. However, this increment operation is not synchronized, leading to a race condition:

**java**

package MultipleThreading;  
  
class Counter  
{  
 int count;  
 // public void increment()  
 public void increment()  
 {  
 count++;  
 }  
}  
public class RaceConditionExample {  
 public static void main(String[] args) throws InterruptedException {  
 Runnable obj1= () ->  
 {  
 for(int i=0;i<=1000;i++){  
 c.increment();  
 }  
 };  
 Runnable obj2= ()-> {  
 for(int i=0;i<=1000;i++){  
 c.increment();  
 }  
 };  
 Thread t1= new Thread(obj1);  
 Thread t2= new Thread(obj2);  
  
 t1.start();  
 t2.start();  
  
 t1.join();  
 t2.join();   
  
 System.*out*.println(c.count);  
  
 }  
}

To fix the race condition, you would need to **synchronize** the **increment**() method to ensure that only one thread can modify the count variable at a time. For example, you could use the **synchronized** keyword or other **synchronization** mechanisms provided by Java to ensure thread safety and prevent race conditions.

package MultipleThreading;  
  
class Counter  
{  
 int count;  
 // public void increment()  
 public synchronized void increment()  
 {  
 count++;  
 }  
}  
public class RaceConditionExample {  
 public static void main(String[] args) throws InterruptedException {  
 Runnable obj1= () ->  
 {  
 for(int i=0;i<=1000;i++){  
 c.increment();  
 }  
 };  
 Runnable obj2= ()-> {  
 for(int i=0;i<=1000;i++){  
 c.increment();  
 }  
 };  
 Thread t1= new Thread(obj1);  
 Thread t2= new Thread(obj2);  
  
 t1.start();  
 t2.start();  
  
 t1.join();  
 t2.join();  
  
 System.*out*.println(c.count);  
  
 }  
}

**Thread States**

In Java, threads can exist in several states throughout their lifecycle. These states represent different stages of a thread's execution and provide insights into what the thread is currently doing. The various thread states in Java are as follows:

1. **New:**

* When a thread is created but has not yet started executing, it is in the New state.
* In this state, the thread has been instantiated but hasn't been started by calling its start() method.

1. **Runnable:**

* A thread that is ready to run is in the Runnable state.
* This means that the thread is eligible to be scheduled by the thread scheduler to run on the CPU.
* Threads in this state may be currently running or waiting for their turn to execute.

1. **Blocked (or Waiting for Monitor Lock):**

* A thread in the Blocked state is waiting for a monitor lock to enter a synchronized block or method.
* This can happen when the thread attempts to enter a synchronized block or method that is already held by another thread.

1. **Waiting (or Timed Waiting):**

* A thread in the Waiting state is waiting indefinitely (or for a specified period) for another thread to perform a particular action.
* This can occur when a thread calls methods like wait(), join(), or sleep().

1. **Timed Waiting:**

* Similar to the Waiting state, but the thread is waiting for a specified period of time.
* This can occur when a thread calls methods like wait(timeout), join(timeout), or sleep(timeout).

1. **Terminated (or Dead):**

* A thread is in the Terminated state when it has completed its execution or terminated abnormally (due to an uncaught exception).
* Once a thread has terminated, it cannot be restarted or run again.

These thread states are important for understanding the behavior and lifecycle of threads in Java applications. By observing and managing thread states effectively, developers can ensure proper synchronization, coordination, and performance of multithreaded programs.