Q : what is a thread in multi threading:

Ans :A thread is a lightweight sub-process, the basic unit of CPU execution. A thread is part of a larger process, and a process can have one or more threads running within it. Each thread shares the same memory space of the process it belongs to, but they run independently.

In simpler terms:

* A **process** is a program that is running.
* A **thread** is a single execution path within that program (process).

Ques: What is process in Multithreading?

Ans: In the context of **multithreading**, the term **process** still refers to an independent program that runs on the operating system, but there is an important distinction to be made between **processes** and **threads**.

**Key Concepts:**

1. **Process**: A process is an independent program that has its own memory space, resources (like file handles, network connections), and other necessary data to execute. It is an isolated unit of execution in the operating system, and multiple processes can run concurrently on the system.
2. **Thread**: A thread, on the other hand, is a smaller unit of execution within a process. Multiple threads can run in parallel within the same process, sharing the same memory and resources of that process.

Task 3:

Extending Thread class

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

ThreadDemo( String name){

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

} catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void Thstart () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

ThreadDemo T1 = new ThreadDemo( "Thread-1");

T1.Thstart();

ThreadDemo T2 = new ThreadDemo( "Thread-2");

T2.Thstart();

}

}

Task 4:

class RunnableDemo implements Runnable{

private Thread t;

private String threadName;

RunnableDemo ( String name){

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

} catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void Thstart () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

RunnableDemo T1 = new RunnableDemo ( "Thread-1");

T1.Thstart();

RunnableDemo T2 = new RunnableDemo ( "Thread-2");

T2.Thstart();

}

}

Task 5 :

class Counter {

private int count = 0;

public void increment() {

count++;

}

public int getCount() {

return count;

}

}

class ThreadDemo extends Thread {

Counter counter;

ThreadDemo(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 10; i++) {

counter.increment();

}

}

}

public class Main {

public static void main(String[] args) {

Counter counter = new Counter();

ThreadDemo t1 = new ThreadDemo(counter);

ThreadDemo t2 = new ThreadDemo(counter);

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final count: " + counter.getCount());

}

}

Output : Final count is 20

Task 7: Static Synchronization:  
Synchronize static methods to ensure only one thread can execute them for the class, not the instance.

class Counter {  
    private static int count = 0;  
  
    public static synchronized void increment() {  
        count++;  
    }  
  
    public static int getCount() {  
        return count;  
    }  
}

class Counter {

private static int count = 0;

// Synchronize the static method to ensure only one thread can modify the static count at a time

public static synchronized void increment() {

count++;

}

public static int getCount() {

return count;

}

}

class ThreadDemo extends Thread {

Counter counter;

// ThreadDemo doesn't need an instance of Counter to access the static method

public void run() {

for (int i = 0; i < 10; i++) {

Counter.increment(); // Static method call, not tied to an instance of Counter

}

}

}

public class Main {

public static void main(String[] args) {

ThreadDemo t1 = new ThreadDemo();

ThreadDemo t2 = new ThreadDemo();

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Static count is shared between all threads

System.out.println("Final count: " + Counter.getCount()); // Should print 20

}

}

Output: Final count: 20

Task 9: Locks:  
Use `java.util.concurrent.locks.Lock` for more sophisticated thread synchronization.

import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
class Counter {  
    private int count = 0;  
    private final Lock lock = new ReentrantLock();  
  
    public void increment() {  
        lock.lock();  
        try {  
            count++;  
        } finally {  
            lock.unlock();  
        }  
    }

public int getCount() {  
        return count;  
    }  
}

Code:

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Counter {

private int count = 0;

private final Lock lock = new ReentrantLock(); // Lock object

// Increment method with explicit lock handling

public void increment() {

lock.lock(); // Acquire the lock

try {

count++; // Critical section (accessing shared resource)

} finally {

lock.unlock(); // Always release the lock, even if an exception occurs

}

}

public int getCount() {

return count;

}

}

class ThreadDemo extends Thread {

Counter counter;

public ThreadDemo(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 10; i++) {

counter.increment();

}

}

}

public class Main {

public static void main(String[] args) {

Counter counter = new Counter();

ThreadDemo t1 = new ThreadDemo(counter);

ThreadDemo t2 = new ThreadDemo(counter);

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final count: " + counter.getCount()); // Should print 20

}

}

Output: Final count: 20

Task 10:

Dead Lock 👍

**Example of Deadlock**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.concurrent.TimeUnit;

class Resource {

private final Lock lock1 = new ReentrantLock();

private final Lock lock2 = new ReentrantLock();

void method1(Resource r) {

try {

// Try acquiring lock1 with timeout

if (lock1.tryLock(100, TimeUnit.MILLISECONDS)) {

try {

System.out.println(Thread.currentThread().getName() + " is executing method1");

try { Thread.sleep(100); } catch (InterruptedException e) {}

// Try acquiring lock2 with timeout

if (r.lock2.tryLock(100, TimeUnit.MILLISECONDS)) {

try {

r.method2(this);

} finally {

r.lock2.unlock(); // Unlock after usage

}

}

} finally {

lock1.unlock(); // Unlock after usage

}

}

} catch (InterruptedException e)

Task 12:

import java.util.stream.\*;

class DoubleColonOp {

    public static void main(String[] args) {

          Stream<String> stream

            = Stream.of("Heelo", "My",

                        "name", "is",

                        "Prasunamba",

                        ".MK");

        stream.forEach(System.out::println);

    }

}

A **daemon thread** in Java is a type of thread that runs in the background and is typically used for tasks that **don't need to prevent the program from terminating**. These are background tasks that support the main work of the application (e.g., garbage collection, background monitoring, etc.).

### Key Characteristics of Daemon Threads:

1. **Runs in the Background**:
   * Daemon threads perform tasks that are not critical to the main operation of the program. For example, the **Garbage Collector** in Java runs as a daemon thread, cleaning up unused objects in the background.
2. **Does Not Block Program Termination**:
   * The most important feature of a daemon thread is that it **does not prevent the JVM from exiting** when all the user (non-daemon) threads finish executing. If only daemon threads remain, the JVM will shut down, even if those threads are still running.
3. **Automatic Termination**:
   * Once all **non-daemon threads** (user threads) complete their work, the JVM **automatically terminates any running daemon threads**. These threads are essentially ignored when it comes to program termination.