Multi Threading:

Task 1:

What is a Process?

Process is the independent unit of execution. It will have its own memory space and system resources.

Task 2:

What is a Thread?

Thread is the smallest unit of a execution within a process. Threads in the same process share memory and resources. Used to perform multiple tasks concurrently within the same application.

Task 3:

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 start ()

{

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 R1 = new RunnableDemo( "Thread-1");

R1.start();

RunnableDemo R2 = new RunnableDemo( "Thread-2");

R2.start();

}

}

Output:

"C:\Program Files (x86)\Java\jdk-1.8\bin\java.exe" "-javaagent:C:\Users\bsjaysre\AppData\Local\JetBrains\IntelliJ IDEA Community Edition 2025.1.2\lib\idea\_rt.jar=52799" -Dfile.encoding=UTF-8 -classpath "C:\Program Files (x86)\Java\jdk-1.8\jre\lib\charsets.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\deploy.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\access-bridge-32.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\cldrdata.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\dnsns.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\jaccess.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\jfxrt.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\localedata.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\nashorn.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\sunec.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\sunjce\_provider.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\sunmscapi.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\sunpkcs11.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\ext\zipfs.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\javaws.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\jce.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\jfr.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\jfxswt.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\jsse.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\management-agent.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\plugin.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\resources.jar;C:\Program Files (x86)\Java\jdk-1.8\jre\lib\rt.jar;C:\Users\bsjaysre\IdeaProjects\Jaysree\out\production\Jaysree" ThreadMain

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-2

Running Thread-1

Thread: Thread-1, 4

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-2, 1

Thread: Thread-1, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

Process finished with exit code 0

Taks 4:

public class ThreadMain {

public static void main(String args[]) {

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

T1.start();

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

T2.start();

}

}

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);

Thread.sleep(50);

}

} catch (InterruptedException e) {

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

}

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

}

public void start ()

{

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

if (t == null)

{

t = new Thread (this, threadName);

t.start ();

}

}

}

Output:

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Running Thread-2

Thread: Thread-1, 4

Thread: Thread-2, 4

Thread: Thread-2, 3

Thread: Thread-1, 3

Thread: Thread-2, 2

Thread: Thread-1, 2

Thread: Thread-2, 1

Thread: Thread-1, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

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: 20

Task 6:

Use synchronized method:

Hint:

1. Synchronized Method:  
Synchronize the entire method to ensure only one thread can execute it at a time.

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

code:

package June20.Task6\_synchronization;  
  
public class Counter {  
 private int count = 0;  
  
 public synchronized void increment() {  
 count++;  
 }  
  
 public int getCount() {  
 return count;  
 }  
  
}

package June20.Task6\_synchronization;  
  
public class ThreadDemo extends Thread {  
 Counter counter;  
  
 ThreadDemo(Counter counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}

package June20.Task6\_synchronization;  
  
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: 20

Process finished with exit code 0

Task 7:

2. Synchronized Block:  
Synchronize a block of code instead of the entire method, providing more control and efficiency.

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

code:

package June20.Task7\_synchronization;  
public class Counter {  
 private int count = 0;  
  
 public void increment() {  
 synchronized (this) {  
 count++;  
 }  
 }  
  
 public int getCount() {  
 return count;  
 }  
}

package June20.Task7\_synchronization;  
  
public class ThreadDemo extends Thread {  
 Counter counter;  
  
 ThreadDemo(Counter counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}

package June20.Task7\_synchronization;  
  
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: 20

Process finished with exit code 0

Task 8:

3. 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;  
    }  
}

code:

package June20.Task8\_synchronization;  
public class Counter {  
 private int count = 0;  
  
 public void increment() {  
 synchronized (this) {  
 count++;  
 }  
 }  
  
 public int getCount() {  
 return count;  
 }  
}

package June20.Task8\_synchronization;  
  
public class ThreadDemo extends Thread {  
 Counter counter;  
  
 ThreadDemo(Counter counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}

package June20.Task8\_synchronization;  
  
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: 20

Process finished with exit code 0

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.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;  
 }  
}

package June20.Task9\_synchronization;  
  
public class ThreadDemo extends Thread {  
 Counter counter;  
  
 ThreadDemo(Counter counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}

ackage June20.Task9\_synchronization;  
  
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: 20

Process finished with exit code 0

Task 10:

Dead Lock 👍

**Example of Deadlock**

class Resource {  
    synchronized void method1(Resource r) {  
        System.out.println(Thread.currentThread().getName() + " is executing method1");  
        try { Thread.sleep(100); } catch (InterruptedException e) {}  
        r.method2(this);  
    }  
  
    synchronized void method2(Resource r) {  
        System.out.println(Thread.currentThread().getName() + " is executing method2");  
        try { Thread.sleep(100); } catch (InterruptedException e) {}  
        r.method1(this);  
    }  
}  
  
public class DeadlockExample {  
    public static void main(String[] args) {  
        final Resource r1 = new Resource();  
        final Resource r2 = new Resource();  
  
        Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");  
        Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");  
  
        t1.start();  
        t2.start();  
    }  
}

code:

package June20.Task10\_DeadLock;  
  
public class Resource {  
 synchronized void method1(Resource r) {  
 System.*out*.println(Thread.*currentThread*().getName() + " is executing method1");  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
 r.method2(this);  
 }  
  
 synchronized void method2(Resource r) {  
 System.*out*.println(Thread.*currentThread*().getName() + " is executing method2");  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
 r.method1(this);  
 }  
}

package June20.Task10\_DeadLock;  
  
public class DeadLockExample {  
 public static void main(String[] args) {  
 final Resource r1 = new Resource();  
 final Resource r2 = new Resource();  
  
 Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");  
 Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");  
  
 t1.start();  
 t2.start();  
  
 }  
}

output:

Thread-1 is executing method1

Thread-2 is executing method1