Day 10 - 20th June 2025

1) What is a process?

Process is like running a program, for example opening a chrome browser, it gets its memory allocation, data, and resources. Each process has at least one thread. Multiple processes don’t share memories (unless designed to do so). A program in execution is a process.

2) What is a thread?

A thread is the smallest unit of execution within a process, it is a sequence of instructions that can be executed independently. All threads share the same memory space within a process. MS Word has multiple threads: one displaying text, one spell checker, one auto-saving etc. Threads are lightweight compared to Process

Threads allows a program to perform multiple tasks simultaneously, like downloading a file while you browse a website or running animations while processing user input. A process can consist of multiple threads.

Process is a program under execution whereas a thread is part of process.

3) Task 3

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name){ //constructor with 1 parameter

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*(5000);

}

} 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:

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-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

4) Task 4

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 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[]) {

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

T1.start();

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

T2.start();

}

}

Output:

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-2, 2

Thread: Thread-1, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

5) 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());

}

}

a

Output: Final count: 20

Task 6:

Use synchronized method:

class Counter {

private int count = 0;

public synchronized 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());

}

}

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;

// Using synchronized block instead of synchronized method

public void increment() {

synchronized(this) {

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();

// Optional: Add sleep to make thread interleaving more visible

try {

Thread.*sleep*(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

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());

}

}

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; // Static count variable

// Static synchronized method

public static synchronized void increment() {

*count*++;

// Optional: Add print statement to see thread execution

System.*out*.println("Thread " + Thread.*currentThread*().getName() +

" incrementing count to " + *count*);

}

public static 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*(); // Calling static method

try {

Thread.*sleep*(100); // Adding delay to make thread interleaving visible

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class StaticSynchronizationDemo {

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:

Thread Thread-0 incrementing count to 1

Thread Thread-1 incrementing count to 2

Thread Thread-1 incrementing count to 3

Thread Thread-0 incrementing count to 4

Thread Thread-0 incrementing count to 5

Thread Thread-1 incrementing count to 6

Thread Thread-0 incrementing count to 7

Thread Thread-1 incrementing count to 8

Thread Thread-0 incrementing count to 9

Thread Thread-1 incrementing count to 10

Thread Thread-0 incrementing count to 11

Thread Thread-1 incrementing count to 12

Thread Thread-0 incrementing count to 13

Thread Thread-1 incrementing count to 14

Thread Thread-0 incrementing count to 15

Thread Thread-1 incrementing count to 16

Thread Thread-0 incrementing count to 17

Thread Thread-1 incrementing count to 18

Thread Thread-0 incrementing count to 19

Thread Thread-1 incrementing count to 20

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(); // Creating a ReentrantLock

public void increment() {

lock.lock(); // Acquiring the lock

try {

count++;

// Optional: Add print statement to see thread execution

System.*out*.println("Thread " + Thread.*currentThread*().getName() +

" incrementing count to " + count);

} finally {

lock.unlock(); // Releasing the lock in finally block

}

}

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();

try {

Thread.*sleep*(100); // Adding delay to make thread interleaving visible

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class LockDemo {

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:

Thread Thread-0 incrementing count to 1

Thread Thread-1 incrementing count to 2

Thread Thread-1 incrementing count to 3

Thread Thread-0 incrementing count to 4

Thread Thread-0 incrementing count to 5

Thread Thread-1 incrementing count to 6

Thread Thread-1 incrementing count to 7

Thread Thread-0 incrementing count to 8

Thread Thread-1 incrementing count to 9

Thread Thread-0 incrementing count to 10

Thread Thread-0 incrementing count to 11

Thread Thread-1 incrementing count to 12

Thread Thread-0 incrementing count to 13

Thread Thread-1 incrementing count to 14

Thread Thread-0 incrementing count to 15

Thread Thread-1 incrementing count to 16

Thread Thread-1 incrementing count to 17

Thread Thread-0 incrementing count to 18

Thread Thread-1 incrementing count to 19

Thread Thread-0 incrementing count to 20

Final count: 20

Task 10:

Dead Lock 👍

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); // Trying to acquire lock on another resource

}

synchronized void method2(Resource r) {

System.*out*.println(Thread.*currentThread*().getName() + " is executing method2");

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

r.method1(this); // Trying to acquire lock on another resource

}

}

public class LockDemo {

public static void main(String[] args) {

final Resource r1 = new Resource();

final Resource r2 = new Resource();

// Using Lambda expressions for thread creation

Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");

Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");

t1.start();

t2.start();

}

}

Deadlock Explanation:

Thread-1 acquires lock on r1 and tries to get lock on r2

Thread-2 acquires lock on r2 and tries to get lock on r1

Both threads wait for each other indefinitely

**ReentrantLock** in Java is a part of the java.util.concurrent package that helps to achieve synchronization more effectively and optimally compared to the traditional Synchronized keyword. It offers features like,

* Timeouts
* Interruptible locks
* More control over Thread Scheduling

These features make it a valuable tool for managing concurrent access to shared resources with greater precision and adaptability.

A **ReentrantLock** allows a thread to acquire the same lock multiple times, which is particularly useful when a thread needs to access a shared resource repeatedly within its execution. It implements the Lock interface, providing greater control over locking compared to synchronized blocks.

* ReentrantLock tracks a "hold count", which:
* Starts at 1 when a thread first locks the resource.
* Each time the thread re-enters the lock, the count is incremented.
* The count is decremented when the lock is released.
* Once the hold count reaches zero, the lock is fully released.

Task 11:

Inter- thread communication…

Example of Inter-thread Communication

class SharedResource {

private boolean ready = false;

synchronized void produce() {

try {

while (ready) {

wait(); // Wait if buffer is full

}

System.*out*.println("Producing...");

ready = true;

notify(); // Notify consumer

} catch (InterruptedException e) {

e.printStackTrace();

}

}

synchronized void consume() {

try {

while (!ready) {

wait(); // Wait if buffer is empty

}

System.*out*.println("Consuming...");

ready = false;

notify(); // Notify producer

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class InterThreadCommunicationExample {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

// Using method reference

Thread producer = new Thread(resource::produce);

Thread consumer = new Thread(resource::consume);

producer.start();

consumer.start();

}

}

Producing...

Consuming...

12) Task 12 (Method References and Stream Operations):

import java.util.stream.\*;

import java.util.List;

import java.util.Arrays;

class DoubleColonOperatorDemo {

// Static method for demonstration

public static void printInUpperCase(String s) {

System.*out*.println(s.toUpperCase());

}

// Instance method for demonstration

public void printWithPrefix(String s) {

System.*out*.println("String is: " + s);

}

public static void main(String[] args) {

// Example 1: Basic Stream with Method Reference

System.*out*.println("Example 1: Basic Stream");

Stream<String> stream1 = Stream.*of*("Hello", "My", "name", "is", "Prasunamba", ".MK");

stream1.forEach(System.*out*::println);

// Example 2: Static Method Reference

System.*out*.println("\nExample 2: Static Method Reference");

Stream<String> stream2 = Stream.*of*("hello", "world");

stream2.forEach(DoubleColonOperatorDemo::*printInUpperCase*);

// Example 3: Instance Method Reference

System.*out*.println("\nExample 3: Instance Method Reference");

DoubleColonOperatorDemo demo = new DoubleColonOperatorDemo();

Stream<String> stream3 = Stream.*of*("Java", "Programming");

stream3.forEach(demo::printWithPrefix);

// Example 4: Method Reference with List

System.*out*.println("\nExample 4: Method Reference with List");

List<String> list = Arrays.*asList*("Apple", "Banana", "Orange");

list.forEach(System.*out*::println);

// Example 5: Method Reference with Map

System.*out*.println("\nExample 5: Method Reference with Map");

Stream<String> stream4 = Stream.*of*("hello", "world");

stream4.map(String::toUpperCase)

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

}

}

Output:

Example 1: Basic Stream

Hello

My

name

is

Prasunamba

.MK

Example 2: Static Method Reference

HELLO

WORLD

Example 3: Instance Method Reference

String is: Java

String is: Programming

Example 4: Method Reference with List

Apple

Banana

Orange

Example 5: Method Reference with Map

HELLO

WORLD

Task 13 (Thread Interruption):

class InterruptibleThread extends Thread {

public void run() {

try {

// Keep running until interrupted

while (!Thread.*currentThread*().isInterrupted()) {

System.*out*.println(Thread.*currentThread*().getName() + " is running");

Thread.*sleep*(100); // Sleep to make interruption more visible

}

} catch (InterruptedException e) {

System.*out*.println(Thread.*currentThread*().getName() + " was interrupted");

return; // Exit the thread

}

}

}

class LongRunningTask implements Runnable {

@Override

public void run() {

try {

// Simulate some long running operation

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

System.*out*.println("Processing iteration: " + i);

Thread.*sleep*(500);

// Check for interruption

if (Thread.*currentThread*().isInterrupted()) {

System.*out*.println("Detected interruption, cleaning up...");

break;

}

}

} catch (InterruptedException e) {

System.*out*.println("Long running task interrupted");

Thread.*currentThread*().interrupt(); // Restore interrupted status

}

}

}

public class ThreadInterruptionDemo {

public static void main(String[] args) {

// Example 1: Basic Thread Interruption

System.*out*.println("Example 1: Basic Thread Interruption");

InterruptibleThread thread1 = new InterruptibleThread();

thread1.start();

try {

Thread.*sleep*(500); // Let the thread run for a while

thread1.interrupt(); // Interrupt the thread

} catch (InterruptedException e) {

e.printStackTrace();

}

// Example 2: Long Running Task Interruption

System.*out*.println("\nExample 2: Long Running Task Interruption");

Thread thread2 = new Thread(new LongRunningTask());

thread2.start();

try {

Thread.*sleep*(2000); // Let the task run for 2 seconds

thread2.interrupt(); // Interrupt the thread

} catch (InterruptedException e) {

e.printStackTrace();

}

// Example 3: Multiple Threads with Interruption

System.*out*.println("\nExample 3: Multiple Threads with Interruption");

Thread[] threads = new Thread[3];

for (int i = 0; i < 3; i++) {

threads[i] = new InterruptibleThread();

threads[i].setName("Thread-" + i);

threads[i].start();

}

try {

Thread.*sleep*(1000); // Let threads run for a while

// Interrupt all threads

for (Thread t : threads) {

t.interrupt();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Example 1: Basic Thread Interruption

Thread-0 is running

Thread-0 is running

Thread-0 is running

Thread-0 is running

Thread-0 is running

Example 2: Long Running Task Interruption

Thread-0 was interrupted

Processing iteration: 0

Processing iteration: 1

Processing iteration: 2

Processing iteration: 3

Example 3: Multiple Threads with Interruption

Long running task interrupted

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-1 is running

Thread-0 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-0 is running

Thread-1 is running

Thread-2 is running

Thread-1 was interrupted

Thread-2 was interrupted

Thread-0 was interrupted

***why not we directly call run() method? And why do we use start() to call the run internally… Hope this helps you Hemanth..***

The run()  method is just an ordinary method (overridden by *you*). As with any other ordinary method and calling it directly will cause the *current thread* to execute run()

Inside start() method will cause the JVM to spawn a new thread and make the newly spawned thread execute run()..

Even if programmatically we are not creating any thread, For every application, O.S will create a default thread to execute its code with CPU.

Calling run method directly will make that run method execute in that main thread given by O.S.

But the intention of creating a thread class is to make sure that run method executes in a different thread. Unless thread manager of O.S / JVM creates a thread, your run method will not get executed in a separate thread. To request O.S/ JVM  to create the separate thread you have to call start() method which will send a request to O.S / JVM to create a thread. Once O.S / JVM creates a thread, then O.S / JVM will automatically call run method of your thread class in that newly created thread context. And hence your purpose of creating a separate thread and executing your run method in a separate thread will be served.

If you call run method directly, then it is like O.S is not creating any thread for you, and default main thread will execute your run method. No point of creating a separate thread class for that!

Hope I am clear. Let me know if you need more explanation to answer your question.

JVM creates threads, internally JVM will have to send a request to thread manager driver of O.S layer to create a new thread in its thred pool.

So gist of the story is

If we want, we can call run()  method, but if we call run method it will run as just a normal Java method. Whereas if we call strat() it JVM  creates a new thread and run method will be executed on that thread. — where multi threading comes in to picture..

Task 20:

public class ThreadTraceExample {

    public static void main(String[] args) {

        method1();

    }

    public static void method1() {

        method2();

    }

    public static void method2() {

        method3();

    }

    public static void method3() {

        StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();

        System.out.println("Thread Stack Trace:");

        // Iterate through the StackTraceElement array and print details

        for (StackTraceElement element : stackTrace) {

            System.out.println("  Class: " + element.getClassName() +

                               ", Method: " + element.getMethodName() +

                               ", Line: " + element.getLineNumber());

        }

    }

}

=======================

How run is called..

Demon threads – done

Synchronisation..  – done

Inter process communication —done

Interrupting a thread.. –done