# **Day 10 - 20 June 2025**

**Document Name:**Day 10 - hmuvvala@ - Hari Gopal Muvvala

### **Task 001: What is a Process?**

A **process** is an independent program that is being executed by the computer.

Whenever we open any application (like Chrome, IntelliJ, or File Explorer), that application becomes a **process**.

Each process:

* Has its own **memory area (RAM space)**
* Runs in its **own environment**
* Has its **own thread(s)** inside it
* Is **independent** from other processes

In Java, the **JVM (Java Virtual Machine)** is a process.

When we run a .java file, it is executed by the JVM, and that JVM itself is treated as a process by the OS.

For example:

When I write java MyProgram, the JVM starts a new process to run my Java code.

Processes are **heavier** than threads because:

* They don’t share memory with other processes
* Communicating between two processes is difficult and slow (needs Inter-Process Communication like sockets or files)

But they are **safe and isolated**, so a crash in one process doesn’t affect others.

### **Task 002: What is a Thread?**

A **thread** is the smallest unit of execution in a program.

It is like a **lightweight sub-program** that runs **inside a process**.

In Java, when we run a program, the **main thread** starts by default.

If we want, we can create **multiple threads** that run **parallelly or concurrently**.

A thread:

* **Shares memory** with other threads of the same process
* Has its own **execution path**, **stack**, and **life cycle**
* Is managed by the **Thread Scheduler** in Java

#### 📌 Real-Life Example:

Imagine I’m watching a YouTube video:

* **Main thread** plays the video
* **Another thread** loads subtitles
* **One more thread** checks for new notifications

All of this happens at once — that’s multithreading in action.

#### 🔍 Why threads are useful in Java?

* For **faster performance** (no waiting for one task to finish)
* For doing **multiple things at once** (like downloading + UI updates)
* For **better CPU usage**

Threads are **lightweight**, meaning they take **less memory** and **less time** to start than processes.

### Task 003: Creating a Thread using Runnable Interface

#### Program:

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);

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:

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-1 exiting.

Thread Thread-2 exiting.

#### Key Points / My Notes:

* Uses Runnable interface to write thread logic separately from thread object.
* start() method creates and starts the thread.
* Thread names help track execution.
* Threads run concurrently; output may interleave.
* sleep() is used to simulate time delay.

### **Task 004: Creating a Thread by Extending Thread Class**

#### Program:

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();

}

}

}

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:

Similar to Task 3 but class extends Thread.

### Task 005: Counter without Synchronization

#### Program:

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: 17 (or 18, 19, 20 — varies)

#### Key Points / My Notes:

* Two threads modify a shared counter object.
* Output is **not always 20** due to race condition.
* No synchronization is used, so count++ is not thread-safe.
* Shows need for synchronized access in multithreaded programs.

### Task 006 – Using synchronized Keyword

#### Modified Method:

public synchronized void increment() {

count++;

}

#### Output:

Final count: 20

#### Key Points / My Notes:

* synchronized makes increment() thread-safe.
* Only one thread at a time can access increment().
* Race condition is avoided; consistent output.
* Simplest form of synchronization in Java.

### Task 007 – Using synchronized Block

#### Modified Method:

public void increment() {

synchronized(this) {

count++;

}

}

#### Output:

Final count: 20

#### Key Points / My Notes:

* Synchronizes only the part of the code that needs locking.
* Offers better performance than synchronizing entire method.
* Lock is applied only to the critical section.

### Task 008 – Static Synchronized Method

#### Modified Code:

class Counter {

private static int count = 0;

public static synchronized void increment() {

count++;

}

public static int getCount() {

return count;

}

}

#### Output:

Final count: 20

#### Key Points / My Notes:

* Synchronization is applied at **class level**, not object level.
* Useful when shared variable is static.
* All threads accessing Counter.increment() are synchronized on Counter.class.

### Task 009 – Using ReentrantLock

#### Modified Method:

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;

}

}

#### Output:

Final count: 20

#### Key Points / My Notes:

* ReentrantLock provides manual control over lock/unlock.
* Must always release the lock using finally.
* Alternative to synchronized; supports more advanced locking.
* Safe and flexible for complex thread operations.

### Task 010 – Deadlock Scenario

#### Program:

class Resource {

public synchronized void methodA(Resource r2) {

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

try { Thread.sleep(1000); } catch (InterruptedException e) {}

System.out.println(Thread.currentThread().getName() + " trying to call methodB");

r2.methodB(this);

}

public synchronized void methodB(Resource r1) {

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

try { Thread.sleep(1000); } catch (InterruptedException e) {}

System.out.println(Thread.currentThread().getName() + " trying to call methodA");

r1.methodA(this);

}

}

public class DeadlockDemo {

public static void main(String[] args) {

Resource r1 = new Resource();

Resource r2 = new Resource();

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

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

t1.start();

t2.start();

}

}

#### Output (Stuck Example):

Thread-1 is executing methodA

Thread-2 is executing methodB

Thread-1 trying to call methodB

Thread-2 trying to call methodA

Both threads hang here — deadlock!

#### Key Points / My Notes:

* Thread-1 locks r1 and waits for r2; Thread-2 locks r2 and waits for r1.
* Both are waiting for each other forever → deadlock.
* Happens due to circular waiting on synchronized resources.
* Can be prevented by consistent locking order or using tryLock.

### Task 011 – Inter-Thread Communication (wait() / notify())

#### Program:

class SharedResource {

boolean ready = false;

synchronized void produce() {

try {

while (ready) wait();

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

ready = true;

notify();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

synchronized void consume() {

try {

while (!ready) wait();

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

ready = false;

notify();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class Task11 {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

Thread producer = new Thread(() -> resource.produce());

Thread consumer = new Thread(() -> resource.consume());

producer.start();

consumer.start();

}

}

#### Output:

Producing...

Consuming...

#### Key Points / My Notes:

* Uses shared object to coordinate threads.
* wait() pauses thread and releases the lock.
* notify() wakes up the waiting thread.
* Useful for producer-consumer communication.

### Task 012 – Stream API & Double Colon (::) Operator

#### Program:

import java.util.stream.Stream;

public class Task12\_DoubleColon {

public static void main(String[] args) {

Stream<String> stream = Stream.of("Hello", "My", "name", "is", "Hari", "Gopal", "Muvvala");

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

}

}

#### Output:

Hello

My

name

is

Hari

Gopal

Muvvala

#### Key Points / My Notes:

* Uses Stream API to process sequence of strings.
* System.out::println is a method reference using ::.
* Shorter alternative to lambda like s -> System.out.println(s).
* Functional and clean approach to print each element.

### Task 013 – Interrupting a Thread

#### Program:

class InterruptibleThread extends Thread {

public void run() {

try {

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

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

Thread.sleep(100);

}

} catch (InterruptedException e) {

System.out.println("Thread was interrupted");

}

}

}

public class InterruptExample {

public static void main(String[] args) {

InterruptibleThread thread = new InterruptibleThread();

thread.start();

try {

Thread.sleep(500);

thread.interrupt();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

#### Output:

Thread is running

Thread is running

Thread is running

Thread was interrupted

#### Key Points / My Notes:

* interrupt() sends a signal to stop a thread.
* isInterrupted() checks if thread was interrupted.
* InterruptedException is thrown during sleep() if interrupted.
* Graceful way to stop a thread instead of killing it.

### Task 014 – Daemon Thread (Concept Only)

#### My Notes:

* A daemon thread runs in the background to support user threads.
* JVM kills all daemon threads automatically once main/user threads finish.
* Use setDaemon(true) **before** starting the thread.
* Common examples: garbage collection, monitoring services.
* Daemon threads do not prevent program from terminating.

### Task 15 – Debugging Tools in Java

#### Explanation:

List of common debugging tools in Java:

* **IntelliJ IDEA Debugger**
* **Eclipse Debugger**
* **VS Code Java Debugger**
* **JDB (Java Debugger)** – command line
* **Log4j / SLF4J** – for detailed logging
* **Chrome DevTools for JS (if using Java + frontend)**

#### Key Points / My Notes:

* Most IDEs like IntelliJ, Eclipse, and VS Code come with built-in visual debuggers.
* Breakpoints, step-in/step-over, watch variables, and call stack views are commonly used features.
* External logging frameworks help track values and flow over time.
* JDB is rarely used directly, but helpful in limited environments like terminals or remote servers.

### Task 16 – Understanding Error Messages

#### Explanation:

There are two main types of errors in Java:

1. **Compile-Time Errors**
   * Happen before program runs
   * Example: missing semicolon, type mismatch, undeclared variable
   * Example message: ';' expected
2. **Run-Time Errors (Exceptions)**
   * Happen during execution
   * Common types:
     + NullPointerException
     + ArrayIndexOutOfBoundsException
     + IOException
     + ArithmeticException

#### Key Points / My Notes:

* Compile-time errors are usually grammar or syntax related.
* Runtime errors occur when the code compiles but fails at execution.
* Understanding error types helps debug faster and avoid similar mistakes.
* Always read the full error line and check the exact line number in the stack trace.

### Task 17 – What is a Stack Trace?

#### Explanation:

A **stack trace** is a report that shows the sequence of method calls leading up to an error.

It helps:

* Identify where the error occurred
* Understand the method call flow
* Trace back from the crashing point to the root method

#### Key Points / My Notes:

* Appears during exceptions and crash reports.
* Helps locate exact method and line where the issue happened.
* Useful for debugging deep method call chains.
* Can be printed manually using e.printStackTrace().

### Task 18 – Calling run() vs start()

#### Program:

class Test extends Thread {

public void run() {

System.out.println("Thread started.");

}

}

public class ThreadExample {

public static void main(String[] args) {

Test t1 = new Test();

t1.run(); // Not t1.start()

}

}

#### Output:

Thread started.

#### Key Points / My Notes:

* Calling run() directly does **not create a new thread**
* It behaves like a **normal method call** on the current thread
* Use start() to run code on a **separate thread**
* start() calls run() internally in a multithreaded way
* Important distinction in multithreading interviews and behavior

### Task 19 – Runnable vs Thread Class Comparison

#### Program:

class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Code executed in a new thread via Runnable.");

}

}

class MyThread extends Thread {

@Override

public void run() {

System.out.println("Code executed in a new thread via Thread extension.");

}

}

public class Task19 {

public static void main(String[] args) {

MyRunnable runnableInstance = new MyRunnable();

MyThread threadInstance = new MyThread();

Thread t1 = new Thread(runnableInstance);

t1.start(); // For MyRunnable

threadInstance.start(); // For MyThread

}

}

#### Output:

Code executed in a new thread via Runnable.

Code executed in a new thread via Thread extension.

#### Key Points / My Notes:

* Runnable: separates logic from thread creation (preferred in most designs)
* Thread class: logic and thread behavior are bundled together
* Both approaches work, but Runnable is more flexible for inheritance and reuse
* start() is mandatory to run in a new thread; run() alone won’t create a separate thread

### Task 20 – Manually Printing Stack Trace

#### Program:

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:");

for (StackTraceElement element : stackTrace) {

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

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

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

}

}

}

#### Output:

Thread Stack Trace:

Class: Thread, Method: getStackTrace, Line: -1

Class: ThreadTraceExample, Method: method3, Line: 17

Class: ThreadTraceExample, Method: method2, Line: 12

Class: ThreadTraceExample, Method: method1, Line: 8

Class: ThreadTraceExample, Method: main, Line: 4

...

#### Key Points / My Notes:

* Thread.currentThread().getStackTrace() returns an array of method call information
* Used for debugging or error logging to trace back execution flow
* Helps identify which methods led to a problem or exception
* Each StackTraceElement gives class name, method name, and line number

### 