| Advance concepts | Collections Framework intro, Streams, File I/O, Multithreading overview |
| --- | --- |
| Troubleshooting | Debugging Tools, Error Messages and Stack Traces, Breakpoints and Code Stepping, Logging for Debugging, Common Bug Patterns, Debugging Strategies, Hands-on Debugging Practice |

**TASK1**

**What is a Process?**

process is a program in execution. Each process has its own memory space, including the heap and stack memory. Consider it as a separate entity entirely. The Operating System manages these processes, deciding when they run and managing their resources.

**TASK2**

**What is a Thread?**

A thread, on the other hand, is the smallest unit of execution within a process. If a process is a program in action, a thread is a pathway through which the program runs. A single process can have multiple threads, all sharing the same memory space. This shared memory model enables threads to communicate with each other more easily compared to processes.

Thread thread = new Thread(){

public void run(){

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

}

};

thread.start();

**TASK3 - Creating a Thread using Runnable Interface**

**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 Day10\_Task1 {**

**public static void main(String args[]) {**

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

**T1.Thstart();**

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

**T2.Thstart();**

**}**

**}**

**Output**

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-2

Running Thread-1

Thread: Thread-2, 4

Thread: Thread-1, 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.

#### **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.

**TASK4 - Creating a Thread using 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 Day10\_Task2 {**

**public static void main(String args[]) {**

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

**T1.Thstart();**

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

**T2.Thstart();**

**}**

**}**

**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.**

**TASK5 - Counter without Synchronization**

**class Counter {**

**private int count = 0;**

**public void increment() {**

**count++;**

**}**

**public int getCount() {**

**return count;**

**}**

**}**

**class ThreadDemo1 extends Thread {**

**Counter counter;**

**ThreadDemo1(Counter counter) {**

**this.counter = counter;**

**}**

**public void run() {**

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

**counter.increment();**

**}**

**}**

**}**

**public class Day10\_TASK5 {**

**public static void main(String[] args) {**

**Counter counter = new Counter();**

**ThreadDemo1 t1 = new ThreadDemo1(counter);**

**ThreadDemo1 t2 = new ThreadDemo1(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)**

**Final count: 20**

#### **Key Points :**

**●** 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.

**TASK6 - Using synchronized Keyword**

**class Counter1 {**

**private int count = 0;**

**public synchronized void increment() {**

**count++;**

**}**

**public int getCount() {**

**return count;**

**}**

**}**

**class ThreadDemo2 extends Thread {**

**Counter counter;**

**ThreadDemo2(Counter counter) {**

**this.counter = counter;**

**}**

**public void run() {**

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

**counter.increment();**

**}**

**}**

**}**

**public class Day10\_TASK6 {**

**public static void main(String[] args) {**

**Counter counter = new Counter();**

**ThreadDemo2 t1 = new ThreadDemo2(counter);**

**ThreadDemo2 t2 = new ThreadDemo2(counter);**

**t1.start();**

**t2.start();**

**try {**

**t1.join();**

**t2.join();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

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

**}**

**Output**

Using the synchronized keyword on the increment() method ensures:

* Mutual exclusion: only one thread can access the method at a time.
* Visibility: changes made by one thread are visible to others.

This approach prevents **race conditions** that could lead to incorrect results (like a final count less than expected).

**Final count: 20**

#### **Key Points :**

● 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.

**TASK7 – Using synchronized Block**

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

class Counter2 {

private int count = 0;

public void increment() {

synchronized (this) {

count++;

}

}

public int getCount() {

return count;

}

}

class ThreadDemo3 extends Thread {

Counter counter;

ThreadDemo3(Counter counter) {

this.counter = counter;

}

public void run() {

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

counter.increment();

}

}

}

public class Day10\_TASK7 {

public static void main(String[] args) {

Counter counter = new Counter();

ThreadDemo3 t1 = new ThreadDemo3(counter);

ThreadDemo3 t2 = new ThreadDemo3(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**

#### **Key Points :**

● 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.

**TASK8 - Static Synchronized Method**

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

class Counter4 {

private static int *count* = 0;

public static synchronized void increment() {

*count*++;

}

public static int getCount() {

return *count*;

}

}

class ThreadDemo4 extends Thread {

Counter counter;

ThreadDemo4(Counter counter) {

this.counter = counter;

}

public void run() {

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

counter.increment();

}

}

}

public class Day10\_TASK8 {

public static void main(String[] args) {

Counter counter = new Counter();

ThreadDemo4 t1 = new ThreadDemo4(counter);

ThreadDemo4 t2 = new ThreadDemo4(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**

#### **Key Points:**

● 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.

**TASK9 - Using ReentrantLock**

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 Counter5{**

**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;**

**}**

**}**

**class ThreadDemo5 extends Thread {**

**Counter counter;**

**ThreadDemo5(Counter counter) {**

**this.counter = counter;**

**}**

**public void run() {**

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

**counter.increment();**

**}**

**}**

**}**

**Output**

**Final count: 20**

#### **Key Points:**

**●** 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.

**TASK10 - Deadlock**

**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 Day10\_TASK10 {**

**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**

**Thread-1 is executing methodA**

**Thread-2 is executing methodB**

**Thread-1 trying to call methodB**

**Thread-2 trying to call methodA**

#### **Key Points :**

**●** 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.

**TASK11 – Inter-Thread Communication (wait() / notify())**

**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 Day10\_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 :**

**●** Uses shared object to coordinate threads.

● wait() pauses thread and releases the lock.

● notify() wakes up the waiting thread.

● Useful for producer-consumer communication.

**TASK12 – Stream API & Double Colon (::) Operator**

**import java.util.stream.\*;**

**import java.util.stream.Stream;**

**class Day10\_TASK12 {**

**public static void main(String[] args) {**

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

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

**}**

**}**

**Output**

**Heelo**

**My**

**name**

**is**

**Prasunamba**

**.MK**

#### **Key Points :**

**●** Uses Stream API to process sequence of strings.

● System.out::println is a method reference using ::

● Shorter alternative to lambda like stream.forEach(str -> System.out.println(str));.

● Functional and clean approach to print each element.

**TASK13 – Interrupting a Thread**

**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 Day10\_TASK13 {**

**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 is running**

**Thread is running**

**Thread was interrupted**

#### **Key Points:**

**●** 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.

**TASK14**

**What are Daemon threads? Explain…**

A Daemon thread in Java is a background thread that does not prevent the JVM from exiting. These threads are typically used for tasks like background housekeeping (e.g., garbage collection, event monitoring) where their termination doesn’t need to block the program from exiting.

When the main thread finishes its execution (or when there are no more non-daemon threads left), the JVM will automatically terminate any remaining daemon threads.

**TASK15 – Debugging Tools in Java**

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:**

**●** 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.

**TASK16 - Understanding Error Messages**

### **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

1. **Run-Time Errors (Exceptions)**

**○** Happen during execution

○ Common types:

■ NullPointerException

■ ArrayIndexOutOfBoundsException

■ IOException

■ ArithmeticException

**TASK17**

**What is Stack trace.. What will it do?**

### **What is a Stack Trace?**

A stack trace is a report that provides information about the call stack at a specific point in time, typically when an exception is thrown in a program. It shows the sequence of method calls that led to the exception, helping developers debug the application by tracing the flow of execution.

### **What Does a Stack Trace Do?**

* It helps identify where an error occurred in the code, particularly which method or class was executing at the time of the exception.
* It lists the method calls (along with their respective line numbers) that were active on the call stack when the exception was thrown.
* It can be used to trace back from the point of failure to the point where the exception originated.

**TASK18**

**class Test extends Thread{**

**public void run(){**

**System.*out*.println("thread started.");**

**}**

**}**

**public class Day10\_TASK18 {**

**public static void main(String args[]){**

**Test t1 = new Test();**

**t1.run(); //not t1.start**

**}**

**}**

**Output**

**thread started.**

#### **Key Points :**

**●** 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

**TASK19 – Runnable vs Thread Class Comparison**

**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 Day10\_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 :**

**●** 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

**TASK20 - Manually Printing Stack Trace**

**public class Day10\_TASK20 {**

**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: java.lang.Thread, Method: getStackTrace, Line: 2166**

**Class: Day10\_TASK20, Method: method3, Line: 15**

**Class: Day10\_TASK20, Method: method2, Line: 11**

**Class: Day10\_TASK20, Method: method1, Line: 7**

**Class: Day10\_TASK20, Method: main, Line: 3**

#### **Key Points :**

**●** 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