**Unit 3:**

**Exception handling and Threading:** Introduction to exception handling, Hierarchy of exception, Usage of try,catch, throw, throws and finally, Built-in and user defined exceptions, Threads, Creating Threads, Thread lifecycle, Concept of multithreading.

### Exception Handling in Java

#### What is an Exception?

An exception is an unexpected event that occurs during the execution of a program, disrupting its normal flow. It can happen due to various reasons, such as invalid user input, file not found, network issues, or programming errors. If not handled properly, exceptions can cause the program to terminate abruptly.

* Run time error which will occurs during program execution.

### Exception Hierarchy in Java

In Java, exceptions are organized in a hierarchy. The root class of all exceptions is Throwable. Here’s the simplified hierarchy:

Throwable

├── Error (e.g., OutOfMemoryError, StackOverflowError)

└── Exception

├── RuntimeException (Unchecked Exceptions, e.g., NullPointerException, ArithmeticException)

└── Checked Exceptions (e.g., IOException, SQLException)

* **Error**: Represents serious issues that are not expected to be handled by the application (e.g., out of memory,JVM crash,etc).  
  these not handled by the programmer,these are caused by the computers and our softwares.
* **Exception**: Represents issues that can be handled by the program.
  + **Checked Exceptions**: Must be handled at compile time (e.g., file not found).checked at compile time
  + **Unchecked Exceptions**: Occur at runtime and do not need to be explicitly handled (e.g., division by zero).
  + 4main types of exception

1. ArithmeticException-Division by Zero
2. ArrayOutOfBoundsException
3. NumberFormatException
4. NullPointerException.

### Difference Between Error and Exception

| **Aspect** | **Error** | **Exception** |
| --- | --- | --- |
| Type | Serious issues (e.g., JVM running out of memory). | Issues that can be handled by the program (e.g., file not found). |
| Recovery | Cannot be recovered. | Can be recovered using exception handling. |
| Examples | OutOfMemoryError, StackOverflowError. | NullPointerException, IOException, ArithmeticException. |
| Handling | Not expected to be handled by the program. | Must be handled using try-catch or throws. |

## ****Built-in vs User-defined Exceptions in Java****

### ****1. Built-in Exceptions****

Built-in exceptions are predefined exceptions in Java, part of the java.lang package. They are categorized into:

* + **Checked Exceptions**
  + **Unchecked Exceptions** (Runtime)

#### ****Example: Handling a Built-in Exception****

Here’s an example demonstrating **ArithmeticException**:

public class BuiltInExceptionExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // Causes ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero is not allowed.");

}

}

}

**Output:**

Error: Division by zero is not allowed.

### ****2. User-defined Exceptions****

User-defined exceptions are custom exceptions created by extending the Exception or RuntimeException class.

#### ****Example: Creating a Custom Exception****

Here’s an example of a user-defined exception for age validation:

// Custom Exception Class

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

// Main Class

public class UserDefinedExceptionExample {

static void checkAge(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or above to vote.");

}

System.out.println("You are eligible to vote.");

}

public static void main(String[] args) {

try {

checkAge(16); // Will throw an exception

} catch (InvalidAgeException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Age must be 18 or above to vote.

### ****Key Differences****

| **Feature** | **Built-in Exceptions** | **User-defined Exceptions** |
| --- | --- | --- |
| **Definition** | Predefined in Java API | Created by the user |
| **Handling** | Directly used with try-catch | Needs custom class extending Exception |
| **Examples** | NullPointerException, IOException | InvalidAgeException, InsufficientBalanceException |

### Exception Handling

Exception handling is a mechanism to handle runtime errors gracefully so that the program can continue running or terminate properly.

#### How to Handle Exceptions?

Java provides the following keywords and blocks to handle exceptions:

1. **try Block**: Contains code that might throw an exception. It contain statements that may generate exception.
2. **catch Block**: Catches and handles the exception.It contains list of statements that handles exception.

If there is no exception in try block catch block won’t be executed.control goes to the next statement of the catch block.

1. **finally Block**: Executes code regardless of whether an exception occurs or not.
2. **throw Keyword**: Used to implicitly throw an exception to catch block.
3. **throws Keyword**: Declares exceptions that a method might throw.

### Steps to Solve Exception Handling

1. Identify the code that might throw an exception and place it inside a try block.
2. Catch the exception using a catch block and handle it.
3. Use finally to execute cleanup code (e.g., closing files or connections).
4. Use throw to explicitly throw an exception when needed.
5. Use throws in the method signature to declare exceptions that the method might throw.

### Exception Handling Blocks and Keywords

#### 1. try-catch Block

* The try block contains code that might throw an exception.
* The catch block handles the exception.

#### try {

#### int result = 10 / 0; // This will throw an ArithmeticException

#### } catch (ArithmeticException e) {

#### System.out.println("Cannot divide by zero!");

#### }

#### Output:

#### Cannot divide by zero!

#### 2. finally Block

* The finally block always executes, whether an exception occurs or not.
* It is used for cleanup activities like closing files or connections.

#### try {

#### int result = 10 / 0;

#### } catch (ArithmeticException e) {

#### System.out.println("Exception caught!");

#### } finally {

#### System.out.println("This will always execute.");

#### }

#### Output:

#### Exception caught!

#### This will always execute.

#### 3. throw Keyword

* Used to explicitly throw an exception.

#### void checkAge(int age) {

#### if (age < 18) {

#### throw new ArithmeticException("Not eligible to vote!");

#### } else {

#### System.out.println("Eligible to vote!");

#### }

#### }

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

#### checkAge(15); // This will throw an exception

#### }

#### Output:

#### Exception in thread "main" java.lang.ArithmeticException: Not eligible to vote!

#### **User-Defined Exceptions**

**/ Custom exception**

**class InvalidAgeException extends Exception {**

**InvalidAgeException(String message) {**

**super(message);**

**}**

**}**

**// Using the custom exception**

**void validateAge(int age) throws InvalidAgeException {**

**if (age < 18) {**

**throw new InvalidAgeException("Age must be 18 or above!");**

**} else {**

**System.out.println("Valid age!");**

**}**

**}**

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

**try {**

**validateAge(15); // This will throw the custom exception**

**} catch (InvalidAgeException e) {**

**System.out.println(e.getMessage());**

**}**

**}**

**Output:**

Age must be 18 or above!

### Key Points to Remember

1. Use try-catch to handle exceptions.
2. Use finally for cleanup activities.
3. Use throw to explicitly throw exceptions.
4. Use throws to declare exceptions in method signatures.
5. Create custom exceptions by extending the Exception class.

### ****Difference Between**** throw ****and**** throws ****in Java****

Both throw and throws are used for exception handling in Java, but they serve different purposes.

| **Feature** | **throw** | **throws** |
| --- | --- | --- |
| **Purpose** | Used to **manually throw an exception** inside a method. | Declares exceptions that a method **might throw** during execution. |
| **Where is it used?** | Inside a method or block. | In the method signature. |
| **Can throw multiple exceptions?** | No, can only throw **one** exception at a time. | Yes, can declare **multiple** exceptions separated by commas. |
| **Follows with?** | An exception object (e.g., throw new ArithmeticException("Error")) | Exception class names (e.g., throws IOException, SQLException) |
| **Checked/Unchecked Exceptions?** | Can throw both checked and unchecked exceptions. | Only used to declare **checked exceptions**. |

**Program 1: Using throw to Manually Throw an Exception**

import java.util.Scanner;

public class ThrowExample {

public static void findSquareRoot(int number) {

if (number < 0) {

throw new ArithmeticException("Cannot calculate square root of a negative number"); // Manually throwing an exception

}

System.out.println("Square Root: " + Math.sqrt(number));

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a number: ");

int num = scanner.nextInt();

try {

findSquareRoot(num); // Call method that may throw an exception

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

}

scanner.close();

}

}

### ****Output:****

Enter a number: -9

Exception caught: Cannot calculate square root of a negative number

### ****3.Java Program Using**** throws ArithmeticException

public class ThrowsExample {

// Method that declares it may throw an ArithmeticException

public static void divide(int a, int b) throws ArithmeticException {

if (b == 0) {

throw new ArithmeticException("Cannot divide by zero"); // Manually throwing the exception

}

System.out.println("Result: " + (a / b));

}

public static void main(String[] args) {

try {

divide(10, 0); // This will throw an exception

} catch (ArithmeticException e) {

System.out.println("Exception caught in main: " + e.getMessage());

}

}

}

### ****Output:****

Exception caught in main: Cannot divide by zero

Here are Java programs demonstrating the exceptions

### 1. ****ArithmeticException**** (Division by zero)

public class ArithmeticExceptionDemo {

public static void main(String[] args) {

try {

int result = 10 / 0; // Division by zero

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught: " + e.getMessage());

}

}

}

### 2. ****NullPointerException****

public class NullPointerExceptionDemo {

public static void main(String[] args) {

try {

String str = null;

System.out.println(str.length()); // Accessing length of null string

} catch (NullPointerException e) {

System.out.println("NullPointerException caught: " + e.getMessage());

}

}

}

### 3. ****ArrayIndexOutOfBoundsException****

public class ArrayOutOfBoundsExceptionDemo {

public static void main(String[] args) {

try {

int[] arr = {1, 2, 3};

System.out.println(arr[5]); // Accessing out-of-bounds index

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage());

}

}

}

### 4. ****NumberFormatException****

public class NumberFormatExceptionDemo {

public static void main(String[] args) {

try {

int num = Integer.parseInt("abc"); // Trying to parse invalid number

} catch (NumberFormatException e) {

System.out.println("NumberFormatException caught: " + e.getMessage());

}

}

}

### 5. ****Multiple catch blocks handling different exceptions****

public class MultipleCatchDemo {

public static void main(String[] args) {

try {

int a = 10 / 0; // ArithmeticException

int[] arr = new int[5];

System.out.println(arr[10]); // ArrayIndexOutOfBoundsException

String str = null;

System.out.println(str.length()); // NullPointerException

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught: " + e.getMessage());

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage());

} catch (NullPointerException e) {

System.out.println("NullPointerException caught: " + e.getMessage());

} catch (Exception e) {

System.out.println("Some other exception caught: " + e.getMessage());

}

}

}

## ****User-Defined Exception Examples in Java****

User-defined exceptions in Java are created by extending the Exception (checked) or RuntimeException (unchecked) class. Below are **5 different examples** of user-defined exceptions.

### ****InvalidAgeException (Age Validation)****

This exception ensures that a person is eligible to vote (age must be 18+).

// Custom Exception Class

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

// Main Class

public class AgeValidation {

static void validateAge(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or above to vote.");

}

System.out.println("You are eligible to vote.");

}

public static void main(String[] args) {

try {

validateAge(16); // Throws exception

} catch (InvalidAgeException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Age must be 18 or above to vote.

### ****InsufficientBalanceException (Banking System)****

This exception occurs when a user tries to withdraw more money than available.

// Custom Exception Class

class InsufficientBalanceException extends Exception {

public InsufficientBalanceException(String message) {

super(message);

}

}

// Bank Account Class

class BankAccount {

private double balance = 5000;

void withdraw(double amount) throws InsufficientBalanceException {

if (amount > balance) {

throw new InsufficientBalanceException("Insufficient balance! Available: " + balance);

}

balance -= amount;

System.out.println("Withdrawal successful! New balance: " + balance);

}

}

// Main Class

public class BankSystem {

public static void main(String[] args) {

BankAccount account = new BankAccount();

try {

account.withdraw(6000); // Throws exception

} catch (InsufficientBalanceException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Insufficient balance! Available: 5000.0

### ****InvalidMarksException (Student Grade System)****

Throws an exception when marks entered are not within the valid range (0-100).

// Custom Exception Class

class InvalidMarksException extends Exception {

public InvalidMarksException(String message) {

super(message);

}

}

// Student Class

class Student {

void setMarks(int marks) throws InvalidMarksException {

if (marks < 0 || marks > 100) {

throw new InvalidMarksException("Marks must be between 0 and 100.");

}

System.out.println("Marks recorded: " + marks);

}

}

// Main Class

public class StudentGradeSystem {

public static void main(String[] args) {

Student student = new Student();

try {

student.setMarks(110); // Throws exception

} catch (InvalidMarksException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Marks must be between 0 and 100.

### ****ProductOutOfStockException (E-commerce System)****

Throws an exception when a product is not available in stock.

// Custom Exception Class

class ProductOutOfStockException extends Exception {

public ProductOutOfStockException(String message) {

super(message);

}

}

// E-commerce Class

class ECommerce {

private int stock = 0;

void buyProduct() throws ProductOutOfStockException {

if (stock == 0) {

throw new ProductOutOfStockException("Product is out of stock!");

}

stock--;

System.out.println("Product purchased successfully!");

}

}

// Main Class

public class OnlineShopping {

public static void main(String[] args) {

ECommerce store = new ECommerce();

try {

store.buyProduct(); // Throws exception

} catch (ProductOutOfStockException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Product is out of stock!

### ****InvalidUsernameException (Login System)****

This exception occurs when the username doesn't meet criteria.

// Custom Exception Class

class InvalidUsernameException extends Exception {

public InvalidUsernameException(String message) {

super(message);

}

}

// Authentication Class

class Login {

void validateUsername(String username) throws InvalidUsernameException {

if (username.length() < 5) {

throw new InvalidUsernameException("Username must be at least 5 characters long.");

}

System.out.println("Username is valid.");

}

}

// Main Class

public class UserLoginSystem {

public static void main(String[] args) {

Login login = new Login();

try {

login.validateUsername("abc"); // Throws exception

} catch (InvalidUsernameException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}

**Output:**

Exception: Username must be at least 5 characters long.

**Threads**

In Java, a **Thread** is the smallest unit of execution within a program. Threads are lightweight processes that run within the context of a larger process, allowing multiple tasks to be performed simultaneously. Java's multithreading capabilities are built into the **Java Virtual Machine (JVM)**, enabling applications to execute multiple threads concurrently.A **Thread** is a lightweight subprocess that shares the same memory space with other threads in the same application.

The **main thread** is automatically created when a Java program starts.

You can create and manage **multiple threads** to divide work and improve application performance

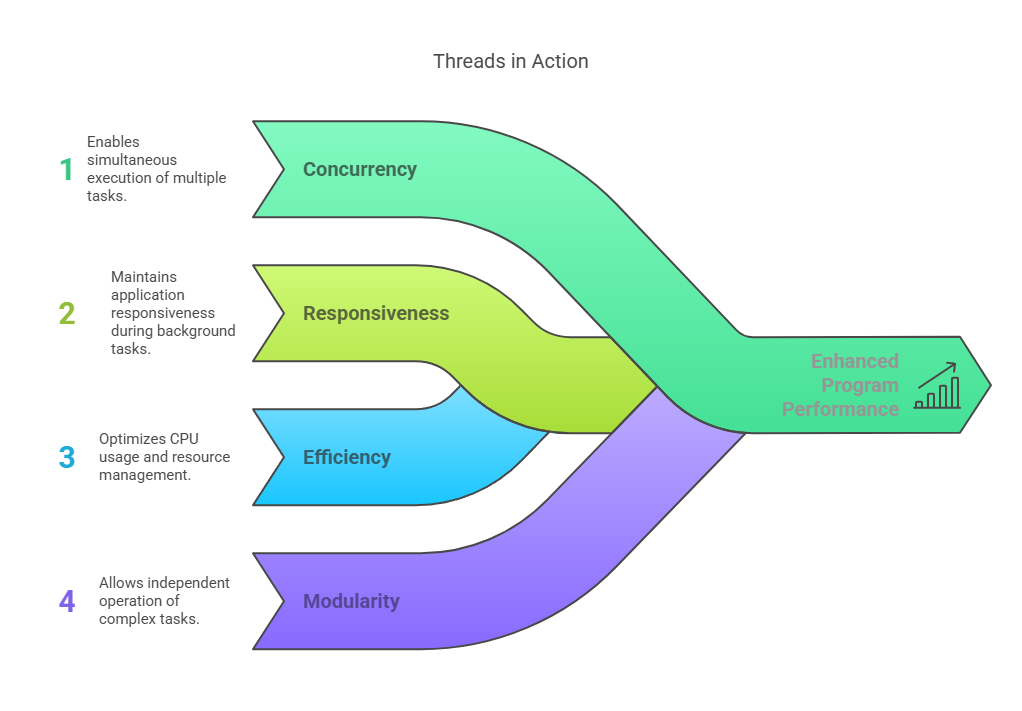
### Why Use Threads?

**Concurrency:** Threads allow a program to do multiple things at the same time.

**Responsiveness:** Background tasks can run without freezing the main application (e.g., downloads).

**Efficiency:** Threads make better use of CPU and system resources by sharing memory within the same process.

**Modularity:** Long or complex operations can be run independently of other program components.



### Key Features of Threads

**Lightweight:** Shares process memory, making them faster to create and switch.

**Independent**: Each thread runs independently, with its own call stack and execution path.

**Concurrent Execution:** Threads can run in parallel (especially on multi-core systems).

## ****Life Cycle of Thread:****

A thread in Java (and most programming languages supporting multithreading) transitions through several states during its lifetime. These states are shown in the diagram and described below:

### ****1. New State****

**Description:** When a thread object is created using the Thread class, it enters the **New** state.

**Code Example:**

Thread t1 = new Thread();

At this point, the thread is just an object—it hasn’t started executing yet.

### ****2. Active State****

This includes **two substates**:

#### ****a) Runnable State****

**Description:** When the start() method is invoked, the thread moves from the New state to the Runnable state.

Here, the thread is **ready to run**, but it **has not yet been assigned CPU time**.

It remains in this state while **waiting for the thread scheduler** to pick it.

#### ****b) Running State****

**Description:** Once the thread scheduler assigns CPU to the thread, it enters the **Running** state.

Now, the thread’s run() method is executed.

Once the time slice is over, it might go back to the Runnable state to wait for another CPU allocation (especially in time-shared systems).

### ****3. Waiting / Blocked State****

A thread enters this state when it is **temporarily paused** and not eligible to run.

This happens when:

**Calling** wait() – the thread waits for a signal (e.g., from another thread).

**Calling** sleep(ms) – the thread sleeps for a given time.

**Calling** suspend() – the thread is suspended manually.

Thread.sleep(1000); // Thread goes to Blocked state for 1 second

A blocked thread moves **back to Runnable** when:

notify() or resume() is called by another thread.

### ****4. Timed Waiting State****

**Description:** This is a state where a thread waits for a specified period of time.

After the time expires, the thread moves back to the Runnable state.

**Common methods that cause this state:**

sleep(time)

wait(time)

join(time)

Thread.sleep()

**Why it's important:** Prevents **starvation** and **deadlock**, ensuring that no thread waits forever without a reason.

### ****5. Terminated / Dead State****

A thread **dies** when:

**It completes execution** (run() method ends).

stop() **method is called** (deprecated, but forcefully stops the thread).

**It encounters an unhandled exception**.

Example:

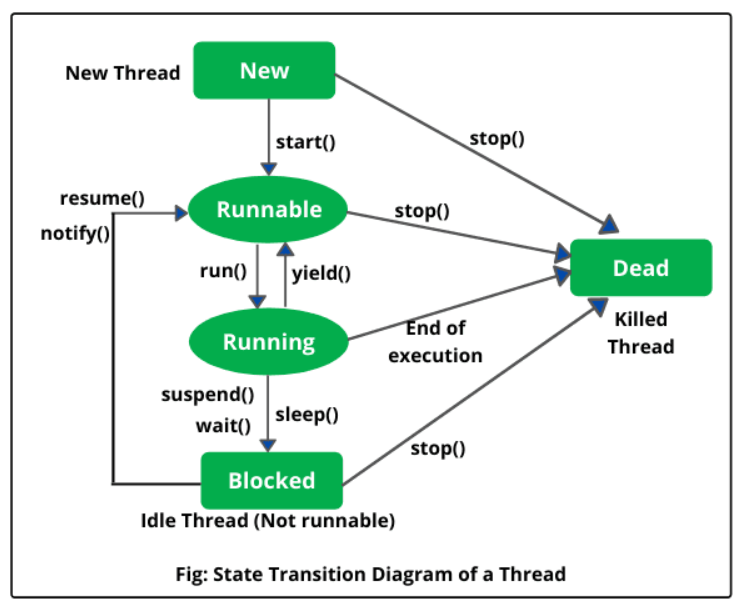
public void run() {

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

} // Thread ends → Moves to Dead state

### ****Transitions Between States****

| **Transition** | **Description** |
| --- | --- |
| **New → Runnable** | Call start() |
| **Runnable → Running** | The CPU scheduler selects the thread |
| **Running → Blocked** | Calls wait(), sleep(), or suspend() |
| **Blocked → Runnable** | Calls notify() or resume() |
| **Running → Dead** | Execution ends or stop() is called |



## ****Thread Class Methods****

1. ****public void run():**** is used to perform action for a thread.
2. ****public void start():**** starts the execution of the thread.JVM calls the run() method on the thread.
3. ****public void sleep(long miliseconds):**** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. ****public void join():**** waits for a thread to die.
5. ****public void join(long miliseconds):**** waits for a thread to die for the specified miliseconds.
6. ****public int getPriority():**** returns the priority of the thread.
7. ****public int setPriority(int priority):**** changes the priority of the thread.
8. ****public String getName():**** returns the name of the thread.
9. ****public void setName(String name):**** changes the name of the thread.
10. ****public Thread currentThread():**** returns the reference of currently executing thread.
11. ****public int getId():**** returns the id of the thread.
12. ****public Thread.State getState():**** returns the state of the thread.
13. ****public boolean isAlive():**** tests if the thread is alive.
14. ****public void yield():**** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. ****public void suspend():**** is used to suspend the thread(depricated).
16. ****public void resume():**** is used to resume the suspended thread(depricated).
17. ****public void stop():**** is used to stop the thread(depricated).
18. ****public void interrupt():**** interrupts the thread.
19. ****public boolean isInterrupted():**** tests if the thread has been interrupted.
20. ****public static boolean interrupted():**** tests if the current thread has been interrupted.

| **Method** | **What it does** |
| --- | --- |
| start() | Starts the thread |
| run() | Code that thread runs |
| sleep() | Pauses thread |
| join() | Waits for thread to finish |
| get/setName() | Get or set thread name |
| get/setPriority() | Get or set thread priority |
| isAlive() | Is the thread still running? |
| yield() | Let other threads run |
| getState() | What's the thread’s current status |
| interrupt() | Try to stop a thread nicely |



# ****How to Create Threads in Java****

Java provides two main ways to create threads:

### 1. ****By Extending the**** Thread ****Class****

You can create a thread by creating a class that extends the Thread class and overriding its run() method.

#### ****Example:****

class MyThread extends Thread {

public void run() {

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

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Start the thread

}

}

**Output:**

Thread Started Running...

### 2. ****By Implementing the**** Runnable ****Interface****

Alternatively, you can implement the Runnable interface and pass the object to a Thread instance.

#### ****Example:****

class MyThread implements Runnable {

public void run() {

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

}

}

public class Main {

public static void main(String[] args) {

Thread t1 = new Thread(new MyThread());

t1.start(); // Start the thread

}

}

**Output:**

Thread is Running Successfully

### Relationship Between start() and run()

When you call start(), it:

Creates a **new thread**.

The new thread internally **calls** run().

So, **you never call** run() **directly**—the JVM does it **after the thread starts**.

### Sequence of Calls

Here’s what happens step by step when you call start():

MyThread t = new MyThread();

t.start(); // You call this

**Explanation:**

JVM creates a new thread.

That new thread is scheduled by the OS.

Inside that thread, the JVM calls run().

Your code in the run() method is executed in the new thread.

So:

start() → creates new thread → JVM internally calls run() → your logic runs

## **Multithreading**

**Multithreading** is a programming concept where **multiple threads** (lightweight units of a process) run **at the same time**, possibly doing different tasks **in parallel or concurrently**.

## Why Use Multithreading?

**Faster performance**: Tasks can run in parallel.

**Better CPU usage**: Modern CPUs have multiple cores.

**Responsive UI**: Keep apps smooth even during heavy tasks (e.g., gaming, downloading, typing).

## Key Terms

| **Term** | **Meaning** |
| --- | --- |
| **Thread** | Smallest unit of execution inside a program |
| **Main Thread** | Default thread where your program starts |
| **Multithreading** | Running multiple threads at the same time |
| **Concurrency** | Tasks appear to run at the same time |
| **Parallelism** | Tasks actually run at the same time (on multiple cores) |

## Java Program: Downloading Files

We'll simulate downloading files using threads and sleep() to mimic time taken.

class DownloadTask extends Thread {

private String fileName;

public DownloadTask(String fileName) {

this.fileName = fileName;

}

public void run() {

System.out.println("Starting download: " + fileName);

try {

Thread.sleep(2000); // Simulating download time

} catch (InterruptedException e) {

System.out.println("Download interrupted: " + fileName);

}

System.out.println("Download complete: " + fileName);

}

}

public class MultiDownloadDemo {

public static void main(String[] args) {

DownloadTask file1 = new DownloadTask("file1.mp4");

DownloadTask file2 = new DownloadTask("file2.pdf");

DownloadTask file3 = new DownloadTask("file3.jpg");

file1.start(); // Starts in a new thread

file2.start(); // Starts in a new thread

file3.start(); // Starts in a new thread

System.out.println("All downloads started...");

}

}

### Output (approximate):

All downloads started…

Starting download: file1.mp4

Starting download: file2.pdf

Starting download: file3.jpg

Download complete: file1.mp4

Download complete: file2.pdf

Download complete: file3.jpg

## **Synchronization**

**Synchronization** is used in **multithreading** to prevent two or more threads from **accessing the same resource (like a variable or method) at the same time**, which could lead to **inconsistent or incorrect results**.

### Why Synchronization is Needed?

When multiple threads share data or resources (like writing to the same file, or updating a balance), they might **interfere** with each other. Synchronization makes sure **only one thread at a time** can access that critical part.

## Java Example: Bank Account Without & With Synchronization

### Without Synchronization

class BankAccount {

int balance = 1000;

void withdraw(int amount) {

if (balance >= amount) {

System.out.println(Thread.currentThread().getName() + " is going to withdraw...");

balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed withdrawal. Remaining: " + balance);

} else {

System.out.println("Insufficient funds for " + Thread.currentThread().getName());

}

}

}

public class NoSyncExample extends Thread {

BankAccount account;

NoSyncExample(BankAccount account) {

this.account = account;

}

public void run() {

account.withdraw(700);

}

public static void main(String[] args) {

BankAccount acc = new BankAccount();

NoSyncExample t1 = new NoSyncExample(acc);

NoSyncExample t2 = new NoSyncExample(acc);

t1.setName("User1");

t2.setName("User2");

t1.start();

t2.start();

}

}

### Problem: Both threads may withdraw at the same time — leading to negative balance.

### With Synchronization

class BankAccount {

int balance = 1000;

synchronized void withdraw(int amount) {

if (balance >= amount) {

System.out.println(Thread.currentThread().getName() + " is going to withdraw...");

balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed withdrawal. Remaining: " + balance);

} else {

System.out.println("Insufficient funds for " + Thread.currentThread().getName());

}

}

}

public class SyncExample extends Thread {

BankAccount account;

SyncExample(BankAccount account) {

this.account = account;

}

public void run() {

account.withdraw(700);

}

public static void main(String[] args) {

BankAccount acc = new BankAccount();

SyncExample t1 = new SyncExample(acc);

SyncExample t2 = new SyncExample(acc);

t1.setName("User1");

t2.setName("User2");

t1.start();

t2.start();

}

}

### Output (Safe):

User1 is going to withdraw...

User1 completed withdrawal. Remaining: 300

Insufficient funds for User2

## 🧠 Summary

| **Term** | **Meaning** |
| --- | --- |
| synchronized | Keyword used to lock a method or block |
| Prevents | Data inconsistency, race conditions |
| Real-life example | ATM queue, printer usage |