Exception

- An **exception** in Java is an **unexpected event** that occurs during the execution of a program, disrupting the normal flow of instructions.
- Java provides a powerful mechanism called **Exception Handling** to handle such situations gracefully.
- An **exception** is an object that represents an error or an unusual condition in a program.
- When an exception occurs, Java creates an **Exception object** and passes it to the **runtime system**.

How Does JVM Handle an Exception?

- When an Exception occurs, **the JVM creates an exception object containing the error name, description and program state. Creating the exception object and handling it in the run-time system is called throwing an exception.**
- **There might** be a list of the methods that had been called to get to the method where an exception occurred. **This ordered list of methods is called call stack.** Now the following procedure will happen:
- The run-time system searches the call stack for an exception handler
- It starts searching from the method where the exception occurred and proceeds backward through the call stack.
- If a handler is found, the exception is passed to it.
- If no handler is found, the default exception handler terminates the program and prints the stack trace.

Types of exceptions:

Checked Exception: These are exceptions that the Java compiler forces one to handle, either by catching them in a try-catch block or by declaring them with the throws keyword in the method signature.

- **Characteristics:** They typically represent recoverable conditions that are outside the immediate control of the program, such as file **I/O errors (IOException), SQL errors (SQLException), or issues with external resources.**

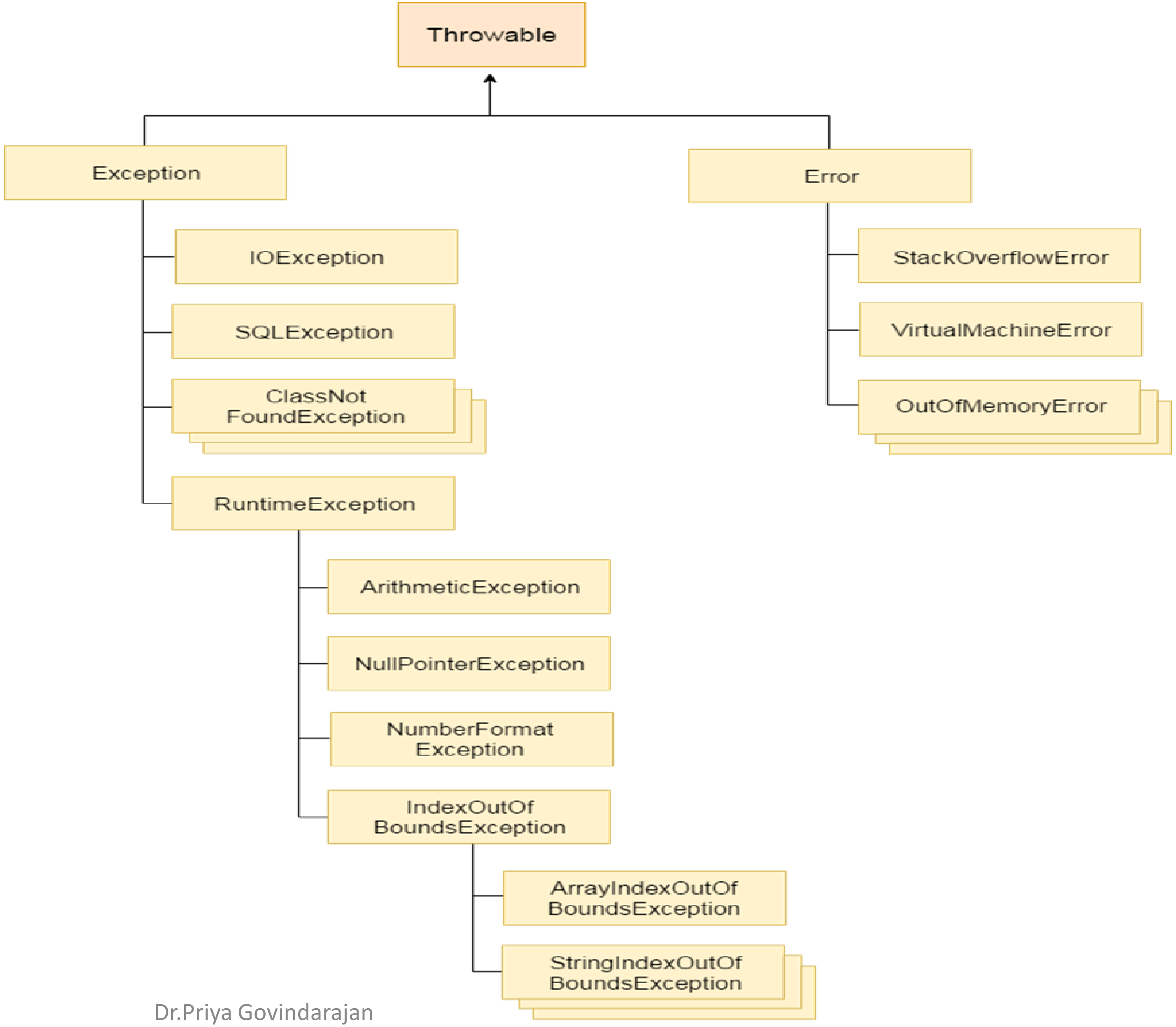
Unchecked Exception: These are exceptions that the compiler does not mandate one to handle. They typically represent programming errors or logical flaws that should ideally be fixed in the code.

- **Characteristics:** They are subclasses of RuntimeException and include common issues like **NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.**

Error: Errors represent serious problems that are usually unrecoverable and indicate a critical issue with the Java Virtual Machine (JVM) or the environment.

- **Characteristics:** They are subclasses of Error and include issues like **OutOfMemoryError or StackOverflowError**. You typically do not catch or handle Errors in your application code, as they signify a fatal state

- **Hierarchy of Java Exception classes:**

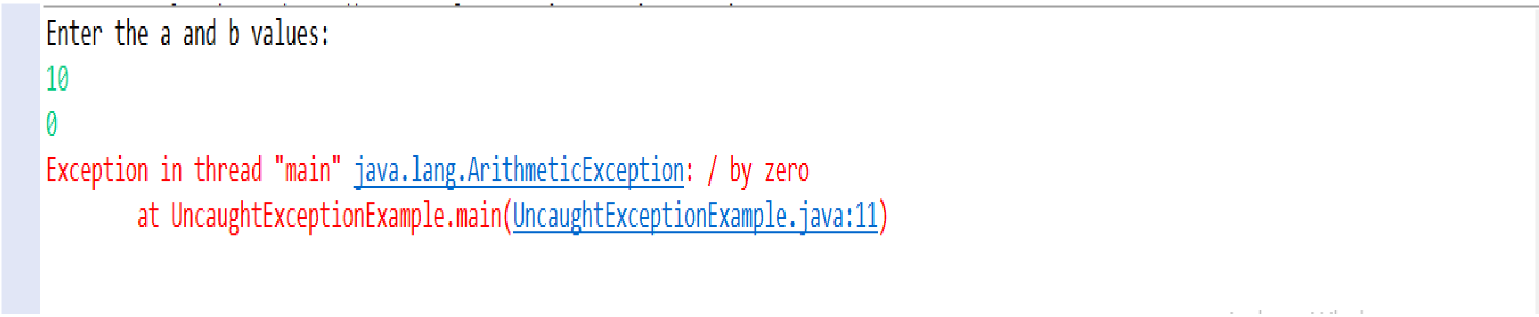


Uncaught Exceptions in Java

1. Example

```
import java.util.Scanner;

public class UncaughtExceptionExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a / b;
        System.out.println(a + "/" + b +" = " + c);

    }
}
```

Output:

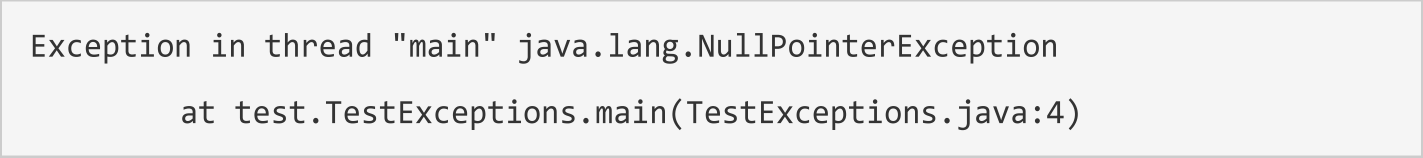


```
Enter the a and b values:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at UncaughtExceptionExample.main(UncaughtExceptionExample.java:11)
```

2. **Example**

```
public class TestExceptions
{
 public static void main(String[] args)
{
   String str = null;
   int len = str.length();
 }
```

we'll get output such as the following:



```
Exception in thread "main" java.lang.NullPointerException
        at test.TestExceptions.main(TestExceptions.java:4)
```

Java Exception Keywords

Keywords
1. try
2. catch
3. finally
4. throw
5. throws

try - catch Blocks:

Syntax of try catch :

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

finally block

syntax :

```
try
{
//code
}
catch (ExceptionType1 e1)
{
  // catch block
}
catch (ExceptionType1 e2)
{
 // catch block
}
finally
{
  // finally block always executes
}
```

Example- 1:

```java
public class TryCatchExample1
{
 public static void main(String[] args)
{
    try
    {
    int data=50/0;
    }
catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

```
Output:
java.lang.ArithmeticException: / by zero
rest of the code
```

Example- 2:

```
public class TryCatchExample2
{
 public static void main(String[] args)
{
    try
    {
    int arr[]= {1,3,5,7};
    System.out.println(arr[10]);
    }
  catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of
bounds for length 4
rest of the code
```

Multiple Catch blocks – Example:

```
class ListOfNumbers
{
  public int[] arrayOfNumbers = new int[10];
  public void writeList()
{
 try {
      arrayOfNumbers[10] = 11;
    }
catch (NumberFormatException e1)
{
      System.out.println("NumberFormatException => " +
e1.getMessage());
    }
catch (IndexOutOfBoundsException e2)
{
      System.out.println("IndexOutOfBoundsException => " +
e2.getMessage());
    }
 }
}
```

```
class Main
{
  public static void main(String[] args)
{
    ListOfNumbers list = new
ListOfNumbers();
    list.writeList();
  }
}
```

```
IndexOutOfBoundsException =>
Index 10 out of bounds for length 10
```

Example : finally block

```
class Main {
  public static void main(String[] args) {
    try {
      String str = "abc";   // not a valid number – if input is “123” – then o/p-?
      int num = Integer.parseInt(str);  // causes NumberFormatException
      System.out.println("Number: " + num);
    }
    catch (NumberFormatException e) {
      System.out.println("NumberFormatException => " + e.getMessage());
    }
    finally {
      System.out.println("Finally block is always executed");
    }
  }
}
```

```
NumberFormatException =>
For input string: "abc"
Finally block is always executed
```

Example :

```
class StringIndexExample {
  public static void main(String[] args) {
    try {
      String str = "Hello";
      System.out.println("Character at index 10: " + str.charAt(10)); // Invalid index
    }
    catch (StringIndexOutOfBoundsException e) {
      System.out.println("Exception caught: " + e);
    }
    finally {
      System.out.println("Finally block executed successfully.");
    }

    System.out.println("Program continues normally...");
  }
}
```

```
Exception caught:
java.lang.StringIndexOutOfBoundsExce
ption: Index 10 out of range for length
5
Finally block executed successfully.
Program continues normally...
```

Nested – Try Statements :

```
class Nest
{
 public static void main(String args[])
{
 //Parent try block
    try{
 //Child try block1
        try{
          System.out.println("Inside block1");
          int b =45/0;
          System.out.println(b);
        }
        catch(ArithmeticException e1){
          System.out.println("Exception: e1");
        }
 //Child try block2
        try{
          System.out.println("Inside block2");
          int b =45/0;
          System.out.println(b);
        }
 catch(ArrayIndexOutOfBoundsException e2)
{
  System.out.println("Exception: e2");
      }
     System.out.println("Just other statement");
   }
catch(ArithmeticException e3)
{
              System.out.println("Arithmetic Exception");
      System.out.println("Inside parent try catch block");
   }
   catch(ArrayIndexOutOfBoundsException e4)
{
             System.out.println("ArrayIndexOutOfBoundsException");
      System.out.println("Inside parent try catch block");
   }
   catch(Exception e5)
{
             System.out.println("Exception");
      System.out.println("Inside parent try catch block");
   }
   System.out.println("Next statement..");
}
 }
```

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try
catch block
Next statement..
```

throw keyword

- **Used to manually throw an exception from the code.**
- **Used inside a method or block.**
- throw a single, specific exception object.

Syntax:

```
throw new ExceptionType("error message");
```

throws keyword

- **Used in a method declaration to specify that the method may throw one or more exceptions**.
- **Used in the method signature**, not inside the method body.
- It tells the caller to **handle or declare** the exception.

Syntax:

```
void myMethod() throws IOException, SQLException
{
    // method code
}
```

Example – throw and throws keywords

```
class Main {
static void checkAge(int age) throws ArithmeticException
{
  if (age < 18) {
    // Manually throw an exception using 'throw'
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  } else {
    System.out.println("Access granted - You are old enough!");
  }
 }
```

```
 public static void main(String[] args) {
   try {
     checkAge(15);  // This will cause an exception
   }
   catch (ArithmeticException e) {
     System.out.println("Exception caught: " + e.getMessage());
   }
   finally {
     System.out.println("Program execution completed.");
   }
  }
}
```

Output:

```
Exception caught: Access denied - You must be at least 18 years old.
Program execution completed.
```

User-Defined /Custom Exception in Java

Java provides us the facility to create our own exceptions by extending the Java Exception class. Creating our own Exception is known as a custom exception in Java or a user-defined exception in Java.

Example 1:

```
class NegativeNumberException extends RuntimeException {
    public NegativeNumberException(String message) {
        super(message);
    }
}
public class RuntimeCustomException {
    static void checkNumber(int num) {
        if (num < 0) {
        throw new NegativeNumberException("Number cannot be negative!");
        }
        System.out.println("Number is valid: " + num);
    }  public static void main(String[] args) {
        checkNumber(-5); // Throws runtime custom exception
    }
}
```

```
Exception in thread "main" NegativeNumberException: Number cannot be negative!
            at RuntimeCustomException.checkNumber(RuntimeCustomException.java:7)
            at RuntimeCustomException.main(RuntimeCustomException.java:12)
```

Example 2:

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message); // Call the constructor of Exception class
    }
}

// Step 2: Use the custom exception in a program
public class UserDefinedExceptionExample {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            // Throwing user-defined exception
            throw new InvalidAgeException("Age is below 18 - Not eligible to vote.");
        } else {
            System.out.println("Eligible to vote!");
        }
    }
```

```
public static void main(String[] args) {
        try {
            checkAge(15); // calling the method
with invalid age
        } catch (InvalidAgeException e) {
            // Handling the user-defined
exception
            System.out.println("Caught
Exception: " + e.getMessage());
        }
        System.out.println("Program continues
normally...");
    }
}
```

Output:

```
Caught Exception: Age is below 18 - Not
eligible to vote.
Program continues normally...
```

IOException

```
import java.io.*;

public class IOExceptionExample2 {
    public static void main(String[] args) {
        try {
            // Create a file and write data to it
            FileWriter writer = new FileWriter("output.txt");
            writer.write("Hello, this is an example of IOException in Java!");
            writer.close();
            System.out.println("File written successfully.");
 // Now, read the file content
            FileReader reader = new FileReader("output.txt");
            BufferedReader br = new BufferedReader(reader);
            String line = br.readLine();
            System.out.println("File content: " + line);
            br.close();

        } catch (IOException e) {
            System.out.println("An IOException occurred: " +
e.getMessage());
        }
    }
}
```

```
File written successfully.
File content: Hello, this is an example of
IOException in Java!
```

ClassNotFoundException

ClassNotFoundException occurs when Java tries to **load a class dynamically** that cannot be found.

```
public class ClassNotFoundExceptionExample {
    public static void main(String[] args) {
        try {
            // Try loading a class that doesn't exist
            Class.forName("com.example.NonExistingClass");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException caught: " + e.getMessage());
        }
    }
}
```

```
ClassNotFoundException caught:
com.example.NonExistingClass
```

Threads

- **In Java, a thread is the smallest unit of execution within a process.**

- **It allows a program to perform multiple tasks concurrently,** improving efficiency and responsiveness — especially in programs involving time-consuming operations like file I/O, network communication, or complex calculations.

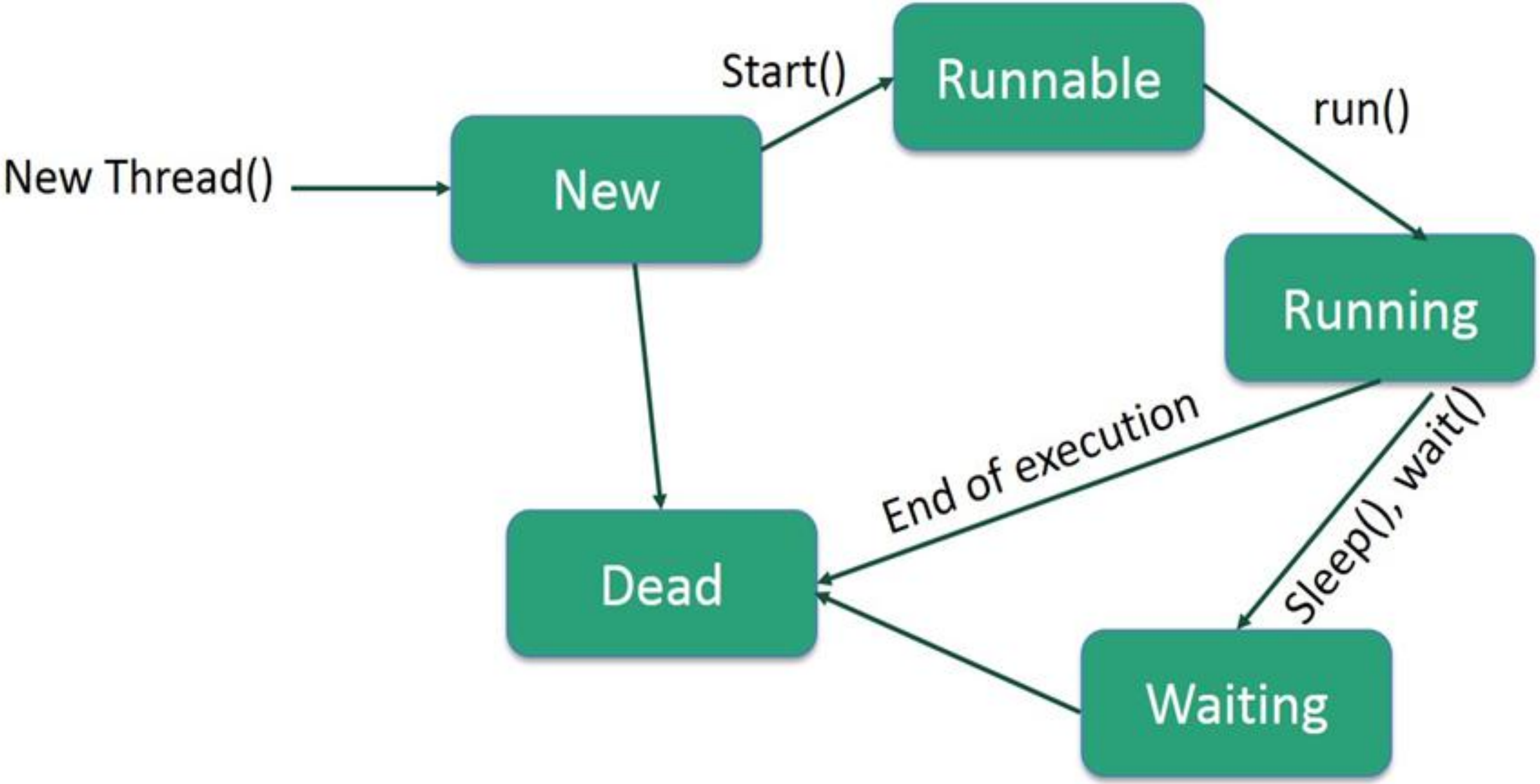
- **A thread represents a path of execution in a program.**

- **Every Java program starts with one thread — the main thread, created automatically by the JVM.**

- **The need for threads in Java (and in programming in general) arises from the requirement to perform multiple tasks simultaneously — improving efficiency, responsiveness, and resource utilization.**

- Example: A text editor can **save files** in the background while the user **types**.

Life Cycle of a Thread



Thread creation in Java

Thread implementation in java can be achieved in two ways:

- Extending the java.lang.Thread class

- Implementing the java.lang.Runnable Interface

1) By extending thread class

Example:

```
public class MyThread extends Thread
{
 public void run()
{
System.out.println("thread is running...");
 }
 public static void main(String[] args)
{
 MyThread obj = new MyThread();
 obj.start();
} }
```

2) By Implementing Runnable interface

Example:

```
public class MyThread implements Runnable
{
  public void run()
{
    System.out.println("thread is running..");
  }
  public static void main(String[] args)
{
    Thread t = new Thread(new MyThread());
    t.start();
} }
```

Multiple threads

```
class MyThread implements Runnable
{
String name;
Thread t;
   MyThread (String thread)
{
   name = threadname;
   t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start();
}
public void run()
{
 try
{
    for(int i = 5; i > 0; i--)
{
    System.out.println(name + ": " + i);
     Thread.sleep(1000);
}
}
```

```
catch (InterruptedException e)
{
    System.out.println(name + "Interrupted");
}
 System.out.println(name + " exiting.");
}
}
class MultiThread
{
public static void main(String args[])
{
    new MyThread("One");
    new MyThread("Two");
    new NewThread("Three");
try {
    Thread.sleep(10000);
}
catch (InterruptedException e)
{
     System.out.println("Main thread Interrupted");
}
     System.out.println("Main thread exiting."); } }
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread:Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Java - Thread Synchronization

- **When multiple threads work together and share the same resource (like a variable, file, or object), problems can occur if they access it simultaneously — this is called a race condition**.
- **To prevent this, Java provides thread synchronization — a mechanism that allows only one thread at a time to access a shared resource.**

Need of thread Synchronization :

- To **prevent data inconsistency** (when multiple threads modify the same data).
- To ensure **thread safety — that shared data remains correct and predictable**.
- To make **critical sections (important code blocks) execute by only one thread at a time**.

Syntax:

```
synchronized(objectidentifier)
{
  // Access shared variables and other shared resources
}
```

Understanding the problem without Synchronization:

```
class Table
{
void printTable(int n)
{  //method not synchronized
   for(int i=1;i<=5;i++)
{
  System.out.println(n*i);
     try
{
      Thread.sleep(400);
     }
catch(Exception e)
{
System.out.println(e);
}
   }
 }
}
```

```
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

```
class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
5
100
10
200
15
300
20
400
25
500
```

Java synchronized method

```
class Table
{
synchronized void printTable(int n)
{  // synchronized  method
   for(int i=1;i<=5;i++)
{
  System.out.println(n*i);
     try
{
      Thread.sleep(400);
     }
catch(Exception e)
{
System.out.println(e);
}
   }
 }
}
```

```
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

```
class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
5
10
15
20
25
100
200
300
400
500
```

Thread Priorities in Java

- In Java, **each thread has a priority**, which helps the **thread scheduler** decide which thread should run first (when multiple threads are waiting for CPU time).
- Thread priority is represented by an **integer value** between **1 and 10**.

Constant	Value	Meaning
`Thread.MIN_PRIORITY`	1	Lowest priority
`Thread.NORM_PRIORITY`	5	Default priority (normal)
`Thread.MAX_PRIORITY`	10	Highest priority

Default priority of any thread = Thread.NORM_PRIORITY (5).

Higher priority threads are **more likely** to get CPU time but not guaranteed.

Thread scheduling is **platform-dependent**.

One can use getPriority() to check a thread's current priority.

Need for Thread Priorities in Java

- When multiple threads are running **simultaneously**, the **CPU (processor)** has to decide **which thread to execute first**.
- Thread priorities help the **thread scheduler** make this decision more intelligently.

Checking Main Thread Priority

```
public class MainThreadPriority {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println("Main thread name: " + t.getName());
        System.out.println("Main thread priority: " + t.getPriority());
    }
}
```

```
Main thread name: main
Main thread priority: 5
```

Demonstrating Thread Priorities

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() +
                           " is running with priority " +
                           Thread.currentThread().getPriority());
    }
}

public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");
```

```
// Set priorities

t1.setPriority(Thread.MIN_PRIORITY);  //
1

t2.setPriority(Thread.NORM_PRIORITY);
// 5

t3.setPriority(Thread.MAX_PRIORITY);  //
10

        // Start threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
Thread-1 is running with priority 1
Thread-3 is running with priority 10
Thread-2 is running with priority 5
```

Example - Showing How Priority May Affect Execution Order

```
class PriorityDemo extends Thread {
  public void run() {
    for (int i = 1; i <= 3; i++) {
      System.out.println(Thread.currentThread().getName() +
                 " - Priority: " + Thread.currentThread().getPriority());
    }
  }
}

public class PriorityTest {
  public static void main(String[] args) {
    PriorityDemo t1 = new PriorityDemo();
    PriorityDemo t2 = new PriorityDemo();
    PriorityDemo t3 = new PriorityDemo();

    t1.setName("LowPriorityThread");
    t2.setName("NormalPriorityThread");
    t3.setName("HighPriorityThread");
```

```
t1.setPriority(3);
    t2.setPriority(5);
    t3.setPriority(8);

    t1.start();
    t2.start();
    t3.start();
  }
}
```

```
LowPriorityThread - Priority: 3
NormalPriorityThread - Priority: 5
HighPriorityThread - Priority: 8
HighPriorityThread - Priority: 8
NormalPriorityThread - Priority: 5
LowPriorityThread - Priority: 3
```

(Execution order still depends on the JVM scheduler — priorities only *suggest* scheduling preferences, not enforce them.)

Need / Purpose of Thread Priorities

Dr.Priya Govindarajan

S.no	Need / Purpose	Explanation
1	**Control over CPU scheduling**	If several threads are ready to run, priorities help the scheduler decide which one should get CPU time first.
2	**Execution of important tasks first**	Some threads may perform more critical operations (e.g., saving user data, processing input). Assigning them higher priority ensures they execute earlier.
3	**Efficient use of system resources**	Helps balance system performance by allowing non-essential threads to run with lower priority.
4	**Avoid starvation of important tasks**	Ensures high-priority tasks are not delayed by less important background processes.
5	**Supports real-time or time-critical operations**	Useful in systems where response time is crucial (like multimedia, simulations, or network servers).

Thread priorities are needed to influence **which thread gets CPU time first**, allowing developers to ensure that **important tasks execute before less important ones** in a multithreaded environment.

Multithreading - Thread-based vs. Process-based Multitasking

1. Process-based Multitasking

Executing multiple programs (processes) at the same time.

Each process has its **own memory space** & runs **independently**.

Example: Running MS Word, a web browser, and a music player at the same time.

Advantages:

One process cannot easily affect another.

Useful for running independent programs.

Disadvantages:

Context switching between processes is **slow**.

Communication between processes is **complex**.

2. Thread-based Multitasking

Executing multiple threads within the same program concurrently.

All threads of a process **share the same memory** but can execute different parts of the code simultaneously.

Example: A text editor performing spell check, saving files, and receiving user input at the same time.

Advantages:

Faster context switching.

Efficient use of CPU time.

Easier data sharing between threads (shared memory).

Disadvantages:

Improper synchronization may cause data inconsistency.

If one thread crashes, it may affect the whole process.

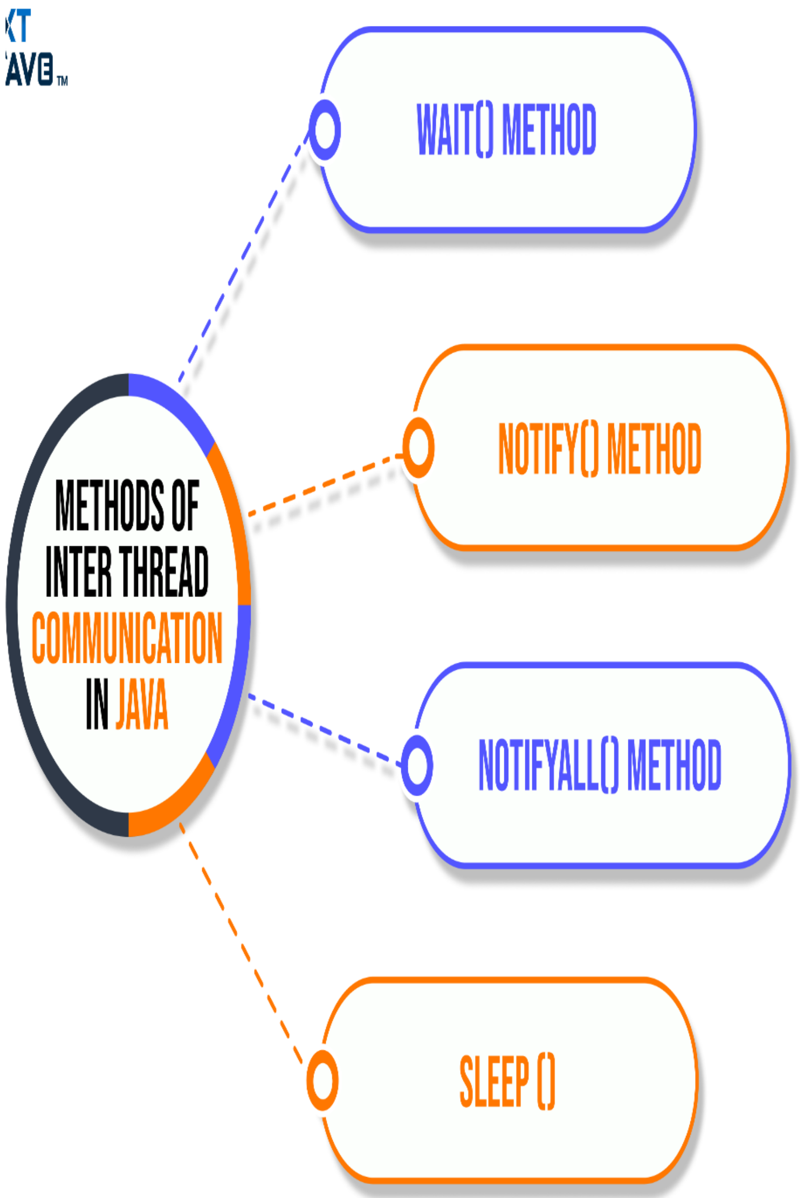
Inter-Thread Communication in Java

- **Inter-thread communication in Java refers to the concepts that helps to synchronize and communicate efficiently between threads that are executing concurrently.** It is a mechanism in which a thread is paused while running in its **critical section and another thread is allowed to enter (or lock) in the same critical section to be executed**. It ensures the communication between the different threads in the same process acts smoothly to coordinate work, share resources, and perform related tasks.

- For a Java multi-threading application, it plays an essential role in coordinating resource management, minimising delays in shared resources, and preventing deadlock and race conditions.

- **Inter-Process Communication (IPC) in Java enables processes to share data and synchronize their actions, whereas, in the context of ITC, attention is directed towards threads within a single Java application.**

- Inter-thread communication synchronizes threads, ensuring proper data sharing or control over execution flow. **While one thread waits for a condition to be fulfilled, it calls wait(), releasing this lock. The other thread can then perform the task and notify the waiting thread using notify() or notifyAll().** The communication ensures that the threads work together and share the resources without race conditions or unnecessary instruction processor utilization.



There are a varieties of methods - implementing inter-thread communication in Java, such as:

1. wait() Method: This method is used when a thread releases the lock it held on a shared resource and goes into a waiting state.

Wait () can be called by a thread only when it finds itself holding the monitor lock of the object on which it is synchronized. It waits for another thread to call, notify or notify all to resume its execution.

wait(long timeout): It will make the thread wait for the specified number of milliseconds before it resumes, even if no notification occurs.

2. notify() Method: This is when the thread has finished its job and wants to wake up another sleeping thread waiting on the same shared resource. The thread calls the notify() method on the object's monitor to inform one of the threads waiting on it and allows that thread to continue execution.

3. notifyAll() Method: The notifyAll() method ensures all the threads waiting on this object's monitor. Thereafter, the threads will compete for the lock.

4. Sleep (): The method sleep() is utilized to stop the execution of the current thread for some amount of time. It does not release any lock the thread has acquired or wait for other threads to signal it.

notifyAll() is a method of the Object class used in **inter-thread communication** (wait/notify mechanism). It is called on a **monitor object** to wake up *all threads* that are waiting on that object's lock.

What notifyAll() Does

Wakes up **all threads** that previously called wait() on the same object.
All awakened threads will now *compete* to acquire the monitor (lock).
Only **one thread at a time** will get the lock and continue execution; others keep waiting.

wait()	sleep()
This belongs to the Object class.	This belongs to the Thread class.
It can only be called inside a synchronized block or method.	It can be called at any time, whether in a synchronized block or not.
This releases the lock on the object so other threads can access the shared resource.	Does not allow for releasing the lock on the object the thread had locked on the whole sleep duration.
The thread will wait until it receives a call notifying it to resume execution.	The thread will halt for the stated time and continue automatically to resume with the time.

Simple Example: Inter-thread Communication with wait(), notify(), notifyAll(), and sleep()

```
class SharedResource {
    synchronized void waitExample() {
        System.out.println(Thread.currentThread().getName() + " is going to wait...");
        try {
            wait(); // Thread waits
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(Thread.currentThread().getName() + " resumed after being notified!");
    }

    synchronized void notifyExample() {
        System.out.println(Thread.currentThread().getName() + " is notifying one waiting thread...");
        notify(); // Wakes up one waiting thread
    }
```

```
 synchronized void notifyAllExample() {
        System.out.println(Thread.currentThread().getName() + " is notifying all waiting threads...");
        notifyAll(); // Wakes up all waiting threads
    }
}
public class ThreadCommunicationDemo
{
   public static void main(String[] args)
{
        SharedResource shared = new SharedResource();
        // Thread 1 waits
        Thread t1 = new Thread(() -> shared.waitExample(), "Thread-1");
        // Thread 2 waits
        Thread t2 = new Thread(() -> shared.waitExample(), "Thread-2");
 Thread notifier = new Thread(() ->      //new Thread(() -> { ... }, "Notifier");
{
            try {
                Thread.sleep(2000); // Give time for t1 and t2 to start waiting
            } catch (InterruptedException e) {
                System.out.println(e);
            }
```

- A **new thread** is created using the Thread class constructor.
  The first argument — () -> { ... } — is a **lambda expression**, which defines what the thread will execute (the code inside run()).

- **So it’s equivalent to:**

```
Thread t = new Thread(new Runnable() {
    public void run() {
        synchronized(shared) {
            shared.notifyAllExample();
        }
    }
});
```

```
synchronized (shared) //The synchronized keyword ensures that the Notifier thread acquires the lock on the shared object (shared).
{
            // Try changing between these two lines to see the difference:
            // shared.notifyExample();   // Notifies only one thread
            shared.notifyAllExample();   // Notifies all waiting threads – Called as part of this program.
         }
      }, "Notifier");

      t1.start();
      t2.start();
      notifier.start();
   }
}
```

```
Output:

Thread-1 is going to wait...
Thread-2 is going to wait...
Notifier is notifying all waiting threads...
Thread-2 resumed after being notified!
Thread-1 resumed after being notified!
```

Example 2:

```
class SharedResource {
    synchronized void waitingMethod() {
        System.out.println(Thread.currentThread().getName() + " is waiting...");
        try {
            wait(); // Thread waits and releases the lock
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " is resumed!");
    }

    synchronized void notifyOneThread() {
        System.out.println(Thread.currentThread().getName() + " is notifying one waiting thread...");
        notify(); // Wakes up one waiting thread
    }
```

e.printStackTrace(); is a **method of the Throwable class** (which Exception inherits from).

It prints the **complete details of an exception** to the console — including:

The **type** of exception (like InterruptedException, NullPointerException, etc.)

The **message** (if any)

The **stack trace** — the exact sequence of method calls that led to the exception.

```
synchronized void notifyAllThreads() {
        System.out.println(Thread.currentThread().getName() + " is notifying all waiting threads...");
        notifyAll(); // Wakes up all waiting threads
    }
}

public class WaitNotifyExample {
    public static void main(String[] args) {
        SharedResource shared = new SharedResource();

        // Thread 1
        Thread t1 = new Thread(() -> shared.waitingMethod(), "Thread-1");

        // Thread 2
        Thread t2 = new Thread(() -> shared.waitingMethod(), "Thread-2");

        // Start both waiting threads
        t1.start();
        t2.start();
// Sleep to ensure both threads start waiting
    try {
      Thread.sleep(2000); // main thread sleeps for 2 seconds
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    // Notifier thread
    Thread notifier = new Thread(() -> {
      synchronized (shared) {
        // Change between these two lines to test the difference
        // shared.notifyOneThread();
        shared.notifyAllThreads();
      }
    }, "Notifier");

    notifier.start();
  }
}
```

Sample Output (when using notifyAll())

```
Thread-1 is waiting...
Thread-2 is waiting...
Notifier is notifying all waiting threads...
Thread-1 is resumed!
Thread-2 is resumed!
```

🧾 **Sample Output (when using notify())**

```
Thread-1 is waiting...
Thread-2 is waiting...
Notifier is notifying one waiting thread...
Thread-1 is resumed!
```

(Only one thread resumes; the other keeps waiting.)

Concurrency - Concurrency Issues (Safety, Liveness, Fairness), Locks and Synchronization

Concurrency issues arise in multithreaded programming when multiple threads access shared resources simultaneously, leading to potential problems like data inconsistency and race conditions. Java provides mechanisms to manage these issues.

Concurrency Issues:

- **Safety:** Ensuring that shared data remains in a consistent and valid state, even when accessed concurrently by multiple threads. **A common safety issue is a race condition, where the outcome of a program depends on the unpredictable timing or interleaving of operations by multiple threads.**

Liveness:

Ensuring that threads can make progress and eventually complete their tasks. Liveness issues include:

- **Deadlock:** Two or more threads are blocked indefinitely, each waiting for a resource held by another.
- **Starvation:** A thread is repeatedly denied access to a shared resource, even though it may become available.
- **Livelock: Threads are actively changing their state in response to other threads, but no thread ever makes progress.**
- **Fairness:** Ensuring that **all threads have a reasonable chance to acquire shared resources and execute their critical sections. An unfair system might consistently favor certain threads over others, leading to starvation for the less favored threads.**

Java offers several mechanisms to address concurrency issues and ensure thread safety:

- synchronized Keyword:
  - **Synchronized Methods:** Declaring a method as synchronized ensures that only one thread can execute that method on a given object at a time. The intrinsic lock (monitor) of the object is acquired by the thread entering the synchronized method and released upon exit.
  - **Synchronized Blocks: Provides more fine-grained control by allowing synchronization on a specific object. A thread must acquire the lock of the specified object before entering the synchronized block**.

Example for Lock :

- java.util.concurrent.locks Package: Provides more flexible and powerful locking mechanisms than the synchronized keyword.
- **Lock Interface (**e.g., ReentrantLock – class) Offers explicit control over locking and unlocking. It provides methods like lock(), unlock(). ReentrantLock is reentrant, meaning a thread can acquire the same lock multiple times.

Example :

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class CounterWithLock {
    private int count = 0;
    private final Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();   // Acquire lock
        try {
            count++;   // Critical section
        } finally {
            lock.unlock(); // Always release lock
        }
    }
    public int getCount() {
        return count;
    }
}
```

```
public class LockExample {
    public static void main(String[] args) throws InterruptedException {
        CounterWithLock counter = new CounterWithLock();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
                counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++)
                counter.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count (using Lock): " +
counter.getCount());
    }
}
```

```
Final Count (using Lock):
2000
```

Thread Pool in Java

- A Thread Pool is a collection of pre-created, reusable threads that are kept ready to perform tasks. Instead of creating a new thread every time you need to run something (which is costly in terms of memory and CPU), a thread pool maintains a fixed number of threads. When a task is submitted:
- If a thread is free, it immediately picks up the task and runs it.
- If all threads are busy, the task waits in a queue until a thread becomes available.
- After finishing a task, the thread does not die. It goes back to the pool and waits for the next task.

Benefits of Thread Pool

- **Better Performance:** Threads are reused instead of being created and destroyed repeatedly.
- **Faster Response Time:** Tasks don’t need to wait for a new thread to be created.
- **Reusability:** Threads remain alive after finishing tasks and are reused for future tasks.
- **Resource Management:** Limits the number of concurrent threads, preventing OutOfMemoryError or CPU overload.

Example – Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
// ExecutorService → Interface that manages a thread pool.Executors
//→ Utility class that creates different types of thread pools.

public class ThreadPoolExample {
   public static void main(String[] args) {
      ExecutorService pool = Executors.newFixedThreadPool(3);
//a fixed thread pool - 3 threads
        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            pool.execute(() -> {
                System.out.println("Task " + taskId + " executed by "
                    + Thread.currentThread().getName());
            });   //Closing the lambda and loop
        }
        pool.shutdown();  // Stop accepting new tasks
   }
}
```

The **Thread Pool program output is NOT fixed**, because threads run **concurrently** and the JVM scheduler decides the order.
But the **pattern** of the output is always like this:

Only **3 threads** will execute all 5 tasks.
Each task prints:

```
Task 1 executed by pool-1-thread-1
Task 2 executed by pool-1-thread-2
Task 3 executed by pool-1-thread-3
Task 4 executed by pool-1-thread-1
Task 5 executed by pool-1-thread-2
```

```
//Task 3 executed by pool-1-thread-2
Task 1 executed by pool-1-thread-1
Task 2 executed by pool-1-thread-3
Task 5 executed by pool-1-thread-2
Task 4 executed by pool-1-thread-3
```

Threads pick tasks depending on scheduling and timing.

Futures and Callable, Fork-Join Parallel Framework

Futures and Callable are programming constructs for asynchronous tasks, while the Fork/Join framework is a specific Java parallel programming framework for tasks that can be recursively split and combined. Futures represent the result of an asynchronous computation, and Callable is an interface for tasks that return a result and can throw an exception. **The Fork/Join framework uses a work-stealing algorithm in its ForkJoinPool to execute these tasks efficiently by breaking large jobs into smaller ones.**

Futures and Callable

Callable: An interface representing a task that returns a result and can throw an exception, unlike Runnable.

Future: An object that represents the result of an asynchronous computation and is returned immediately after a task is submitted. one can use a Future object to check the status of the task and retrieve its result later.

Fork/Join Framework

A parallel programming model based on the **"divide and conquer" strategy. A large task is recursively broken down into smaller subtasks until they are small enough to be solved directly.**

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
//ForkJoinPool (class) = A pool of threads that runs
tasks in parallel
// RecursiveAction (class)- A task that can split
into smaller tasks but returns no value

class PrintTask extends RecursiveAction {
  int start, end;

  public PrintTask(int start, int end) {
    this.start = start;
    this.end = end;
  }
  @Override
protected void compute() {
    // If small enough, print directly
    if (end - start <= 5) {
      for (int i = start; i <= end; i++) {
        System.out.println("Number: " + i);
      }
    }
```

```java
else {
      // Otherwise split into two subtasks
      int mid = (start + end) / 2;

      PrintTask task1 = new PrintTask(start, mid);
      PrintTask task2 = new PrintTask(mid + 1, end);

      task1.fork();       // run task1 in parallel
      task2.compute();    // run task2 in current thread
      task1.join();       // wait for task1 to finish
    }
  }
}
public class SimpleForkJoin {
  public static void main(String[] args) {

    ForkJoinPool pool = new ForkJoinPool();

    // Create a task to print numbers 1 to 20
    PrintTask task = new PrintTask(1, 20);

    pool.invoke(task);   // Start Fork–Join execution – ForkJoinPool starts
executing task by calling its compute() method.
  }  }
```

- The program prints **numbers 1 to 20**.
- But **the order of chunks may vary** because Fork–Join runs parts in *parallel*.
- **The output will look like this (one possible example):**

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10
Number: 11
Number: 12
Number: 13
Number: 14
Number: 15
Number: 16
Number: 17
Number: 18
Number: 19
Number: 20
```

The blocks of printing may appear **interleaved** like this:

```
Number: 11
Number: 12
Number: 13
Number: 14
Number: 15
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 16
Number: 17
Number: 18
Number: 19
Number: 20
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10
```