GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

# OOPs with JAVA (BCS403)

Unit: II

Subject Name : Object Oriented Programming with Java (BCS-403)

( B.Tech 4th Sem)

Dr. Birendra Kr. Saraswat
Associate Professor
Department of IT

**Exception Handling**: The Idea behind Exception, Exceptions & Errors, Types of Exception, Control Flow in Exceptions, JVM Reaction to Exceptions, Use of try, catch, finally, throw, throws in Exception Handling, In-built and User Defined Exceptions, Checked and Un-Checked Exceptions.

**Input/ Output Basics**: Byte Streams and Character Streams, Reading and Writing File in Java.

**Multithreading:** Thread, Thread Life Cycle, Creating Threads, Thread Priorities, Synchronizing Threads, Inter-thread Communication.

# Exceptions in Java

## What are Java Exceptions?

**In Java, Exception** is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

**Exception Handling** in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**Major reasons why an exception Occurs**

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

**Errors** represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

**Difference between Error and Exception**

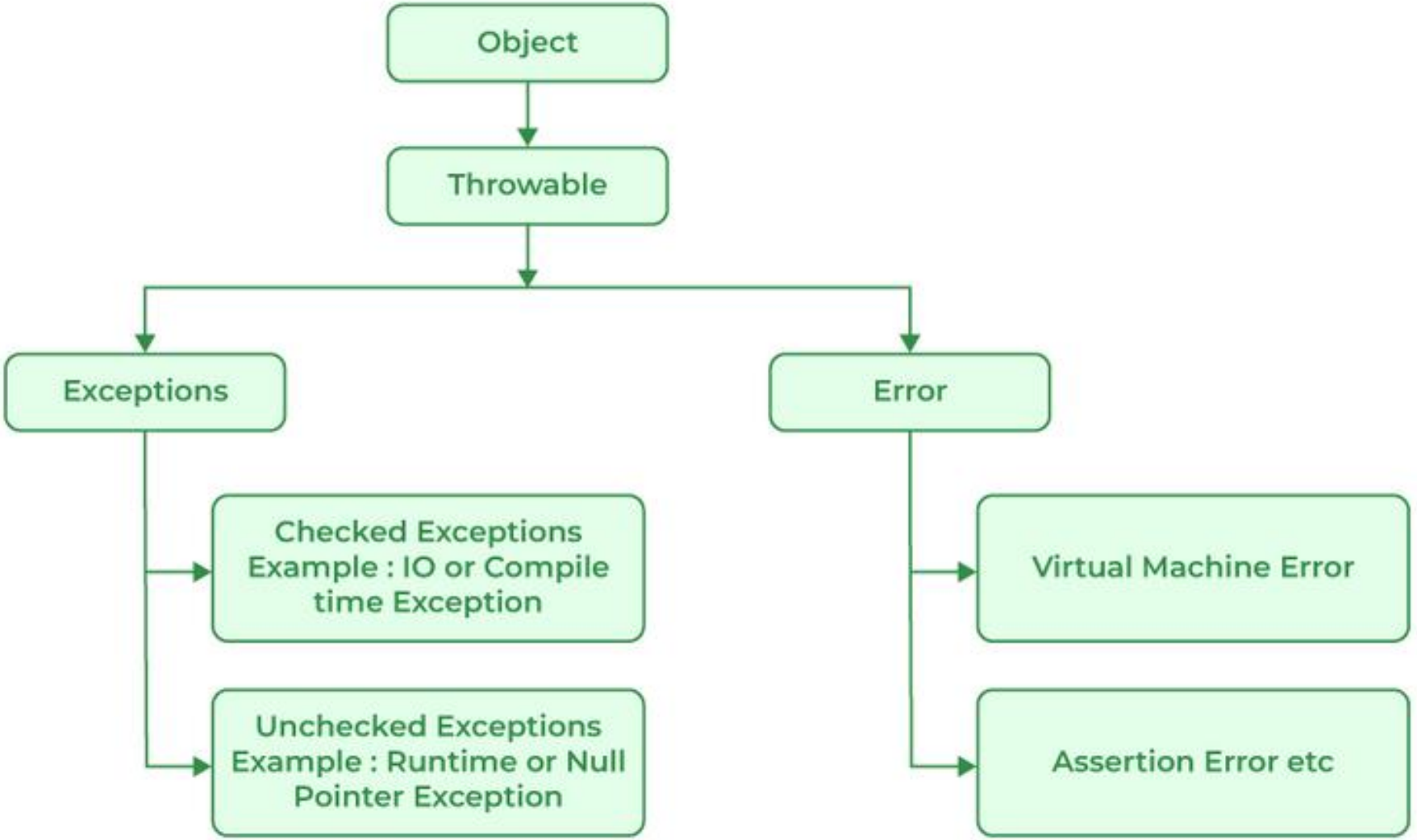
Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

## Exception Hierarchy

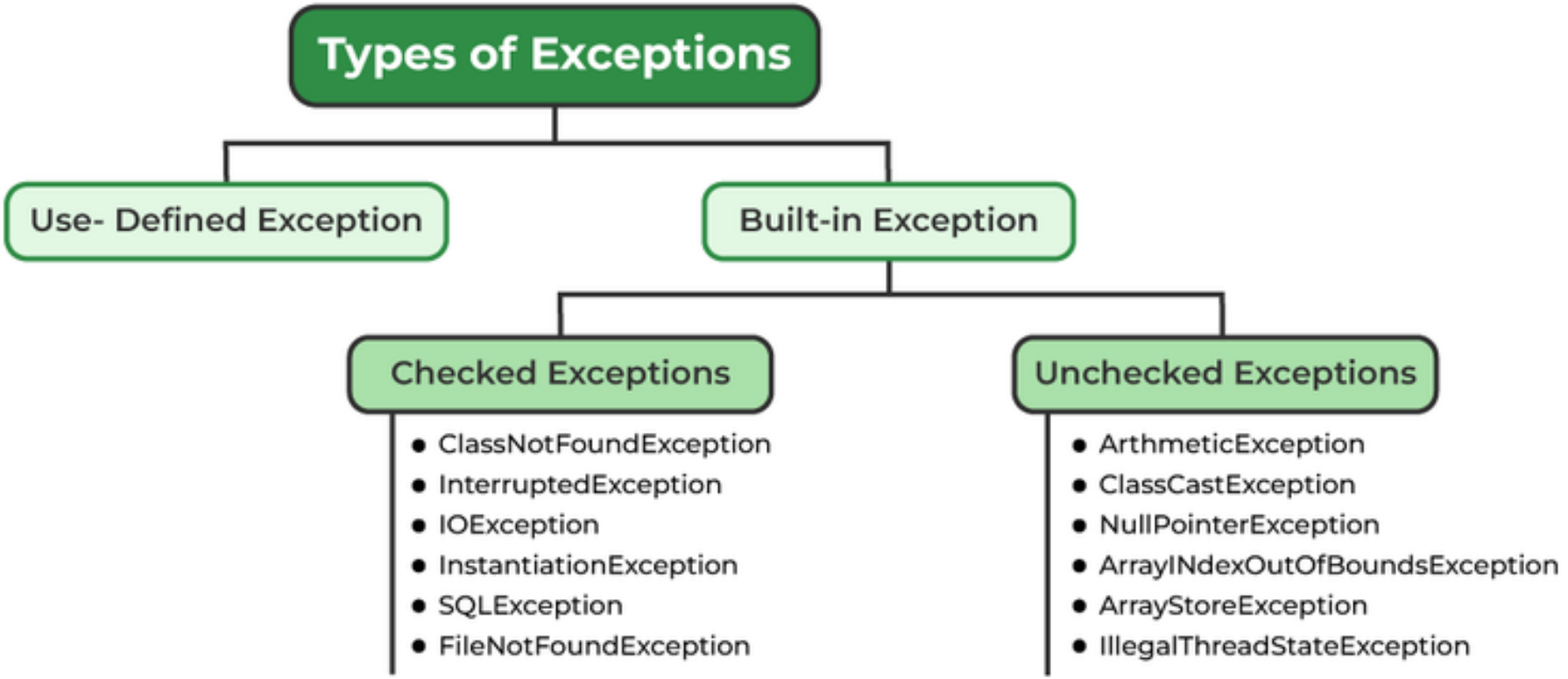
All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.

# Exception Hierarchy



## Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



# Exceptions in Java

## 1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

# Exceptions in Java

## 1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

# Exceptions in Java

## 1. Built-in Exceptions

### Checked Exceptions:

```
import java.io.*;
class CheckedExceptionExample {
    public static void main(String args[]) {
        FileInputStream file_data = null;
        file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
        int m;
        while(( m = file_data.read() ) != -1) {
            System.out.print((char)m);
        }
        file_data.close();
    }
}
```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1.The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.

2.The **read()** method of the **FileInputStream** class throws the **IOException**.

3.The **close()** method also throws the IOException.

# Exceptions in Java

## 1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

# Exceptions in Java

## 1. Built-in Exceptions

### Unchecked Exceptions:

```
class UncheckedException1 {
   public static void main(String args[]
)
   {
    int num[] ={10,20,30,40,50,60};
    System.out.println(num[7]);
   }
}
```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

# Exceptions in Java

**2. User-Defined Exceptions:**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1.Provision to Complete Program Execution
2.Easy Identification of Program Code and Error-Handling Code
3.Propagation of Errors
4.Meaningful Error Reporting
5.Identifying Error Types

## Control Flow in Exceptions

Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. 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.

# Exceptions in Java

•The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.

•The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.

•If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.

•If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.

# Exceptions in Java

## Exception Handling

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Program statements that you think can raise exceptions are contained within a try block.

If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner.

System-generated exceptions are automatically thrown by the Java run-time system.

To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause.

Any code that absolutely must be executed after a try block completes is put in a **finally** block.

# Exceptions in Java

| Keyword | Description |
|---|---|
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

# Exceptions in Java

## Catching Multiple Type of Exceptions

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}
```

# Exceptions in Java

## try and catch

```
try {
   // Protected code
} catch (ExceptionName e1) {
   // Catch block
}
```

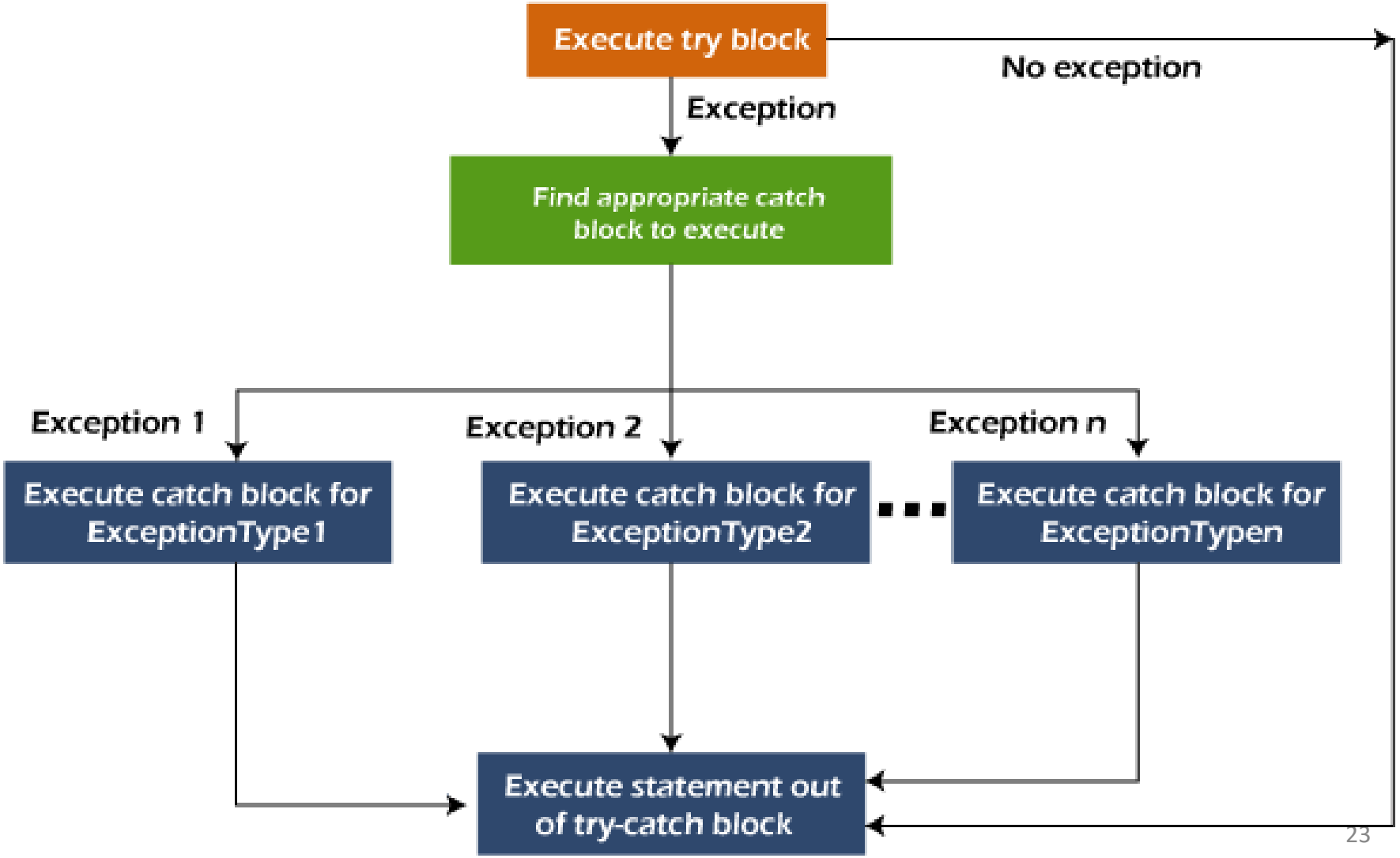
```
public class JavaExceptionExample{

  public static void main(String args[])
{
   try{
      //code that may raise exception
      int data=100/0;
   }
   catch(ArithmeticException e){
      System.out.println(e);
    }
   //rest code of the program
    System.out.println("rest of the code..
.");
   }
}
```

```
public class DivisionExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int a, b;

        try {
            System.out.print("Enter numerator: ");
            a = scanner.nextInt();
            System.out.print("Enter denominator: ");
            b = scanner.nextInt();

            int result = a / b;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero.");
        } finally {
            System.out.println("Division operation completed.");
            scanner.close();
        }
    }
}
```

```
public class ArrayAccessExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};

        try {
            System.out.println("Accessing element at index 5: " + numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Index is out of bounds.");
        } finally {
            System.out.println("Array access operation completed.");
        }
    }
}
```

# Exceptions in Java

## Catching Multiple Type of Exceptions



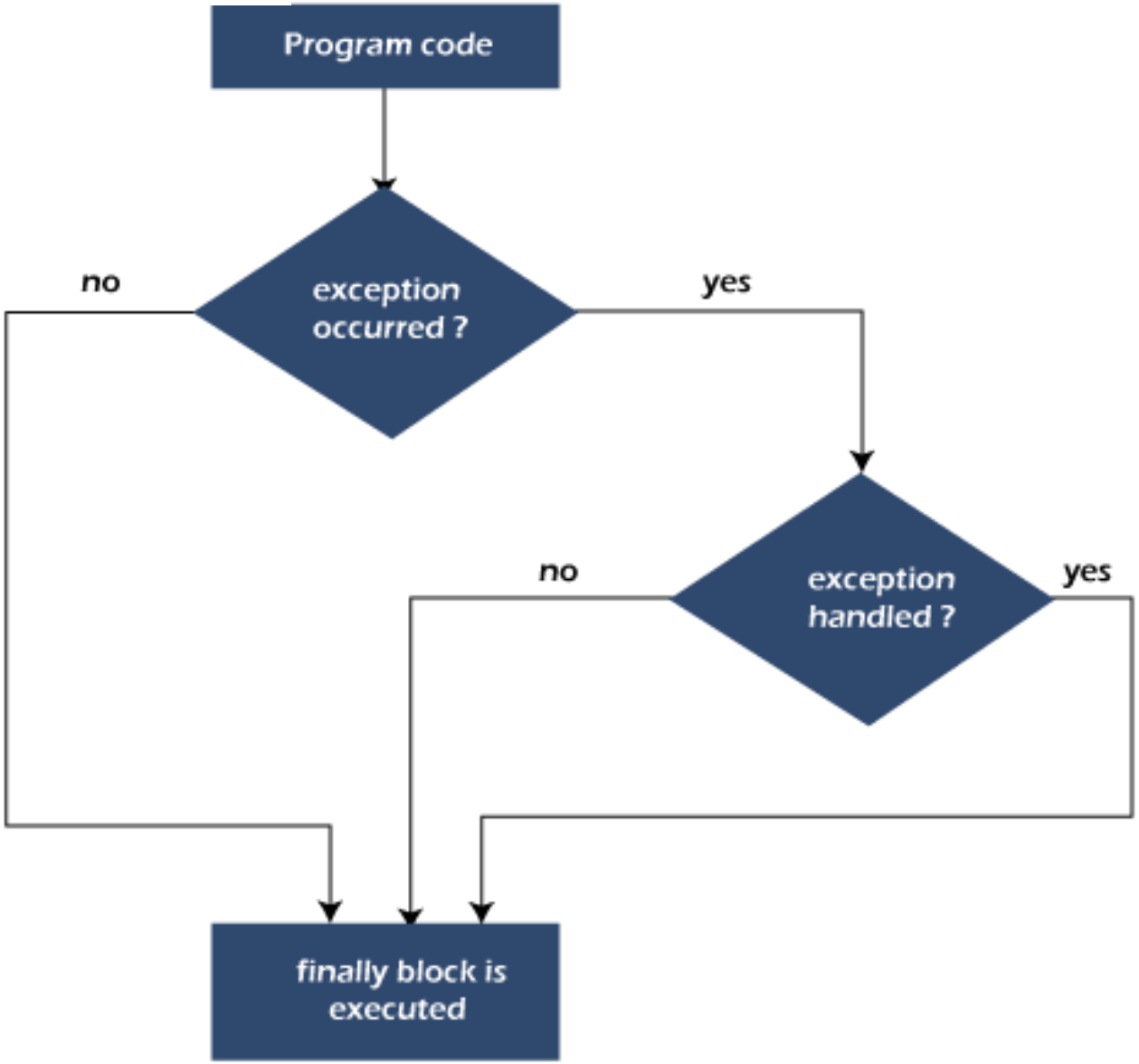
# Exceptions in Java

```
try {
    file = new FileInputStream(fileName);
    x = (byte) file.read();
    } catch (IOException i) {
        i.printStackTrace();
        return -1;
    } catch (FileNotFoundException f) // Not valid! {
        f.printStackTrace();
        return -1;
    }
```

# Exceptions in Java

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

# Exceptions in Java



# Exceptions in Java

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
}
```

## throw and throws in Java

### Java throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

### Syntax in Java throw

```
throw Instance
```

Example: `throw new ArithmeticException("/ by zero");`

But this exception i.e., *Instance* must be of type **Throwable** or a subclass of **Throwable**.

## throw and throws in Java

```
public class TestThrow {
    //defining a method
    public static void checkNum(int num) {
        if (num < 1) {
            throw new ArithmeticException("\nNumber is negative, cannot calc
ulate  square");
        }
        else {
            System.out.println("Square of " + num + " is " + (num*num));
        }
    }
    //main method
    public static void main(String[] args) {
            TestThrow obj = new TestThrow();
            obj.checkNum(-3);
            System.out.println("Rest of the code..");
    }
}
```

# Exceptions in Java

## throw and throws in Java

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

# Exceptions in Java

- Write a java program to create a user defined exception YongerAgeException derived from RuntimeException wich display the message “not eligible for vore” if age is less then 18 years.

```
import java.util.Scanner;

class YoungerAgeException extends RuntimeException
{
    YoungerAgeException(String msg)
    {
        super(msg);
    }
}
class Voting
{
    public static void main(String[] args)
    {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter Your Age : ");

        int age=s.nextInt();
        if(age<18)
        {
            throw new YoungerAgeException("You are not eligible for voting");
        }
        else
        {
            System.out.println("you can vote successfully");
        }
    }
```

# Exceptions in Java

"throws" keyword is used to declare an exception. It gives an information to the caller method that there may occur an exception so it is better for the caller method to provide the exception handling code so that normal flow can be maintained.

# Exceptions in Java

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

class ReadAndWrite
{
    void readFile() throws FileNotFoundException
    {
        FileInputStream fis=new FileInputStream("d:/abc.txt");
    }
}
```

# Exceptions in Java

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

class ReadAndWrite
{
    void readFile() throws FileNotFoundException
    {
        FileInputStream fis=new FileInputStream("d:/abc.txt");
        //statements
    }
    void saveFile() throws FileNotFoundException
    {
        String text="this is demo";
        FileOutputStream fos=new FileOutputStream("d:/xyz.txt");
        //statements
    }
}
```

# Exceptions in Java

```
class Test
{
    public static void main(String[] args)
    {
        ReadAndWrite rw=new ReadAndWrite();
        try
        {
            rw.readFile();
        }
        catch(FileNotFoundException e)
        {
            e.printStackTrace();
        }

    }
}
```

# Exceptions in Java

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

# Exceptions in Java

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try/catch** statement that catches this exception.

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

# Exceptions in Java

## Difference between throw and throws in

| Sr. no. | Basis of Differences | throw | throws |
|---|---|---|---|
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception | Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked |

# Exceptions in Java

## Difference between throw and throws in

| Sr. no. | Basis of Differences | throw | throws |
|---|---|---|---|
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

# Input/Output Basics in Java

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
Java uses the concept of a stream to make I/O operation fast.
The java.io package contains all the classes required for input and output operations.
We can perform **file handling in Java** by Java I/O API.

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

# Input/Output Basics in Java

## Stream

A stream can be defined as a sequence of data. There are two kinds of Streams –

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.

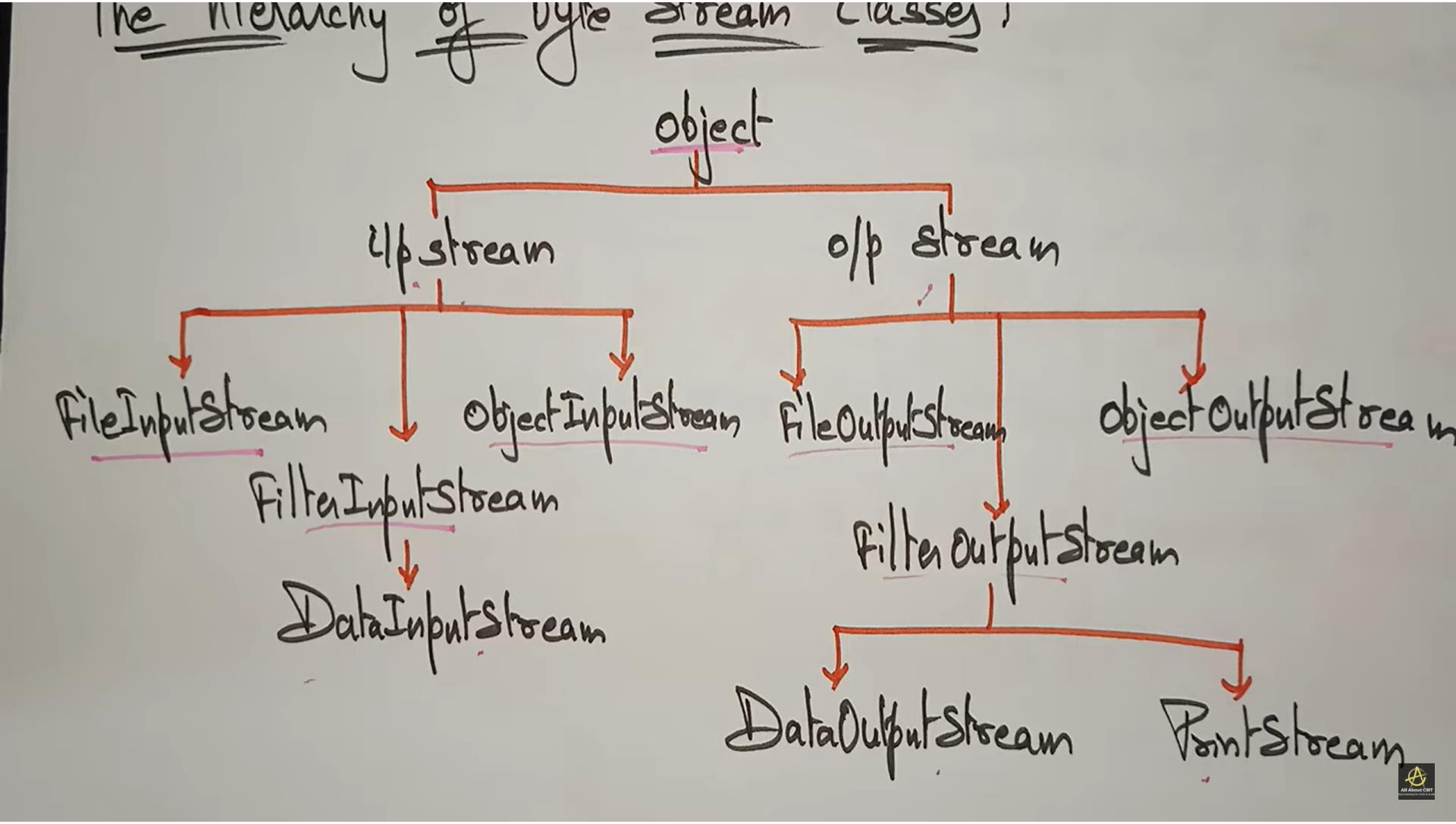
# Input/Output Basics in Java

## Byte Streams

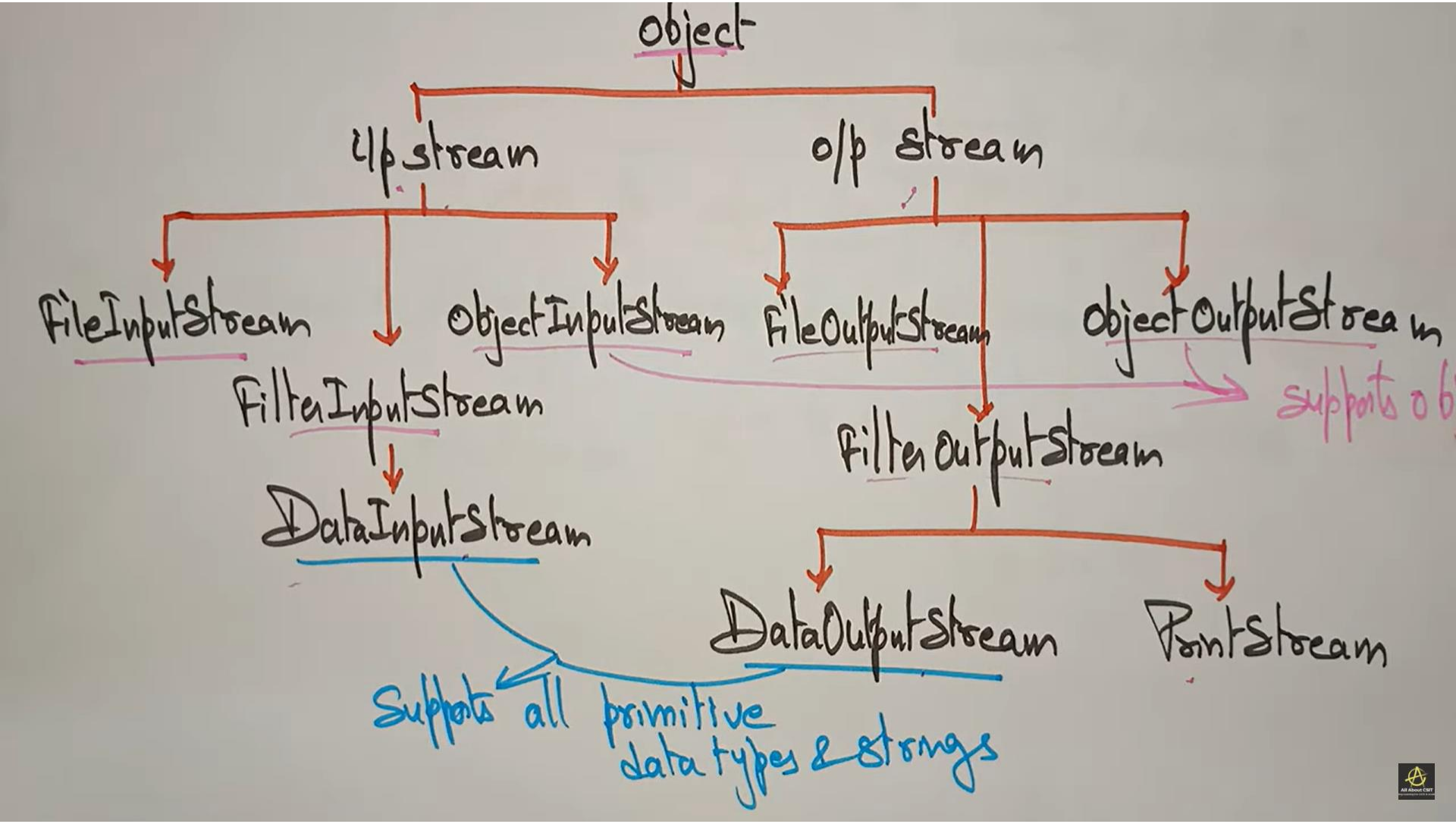
Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream.**

- **Input Stream:** reads data from the source.
- **Output Stream:** writes data to a destination.

# Input/Output Basics in Java



# Input/Output Basics in Java



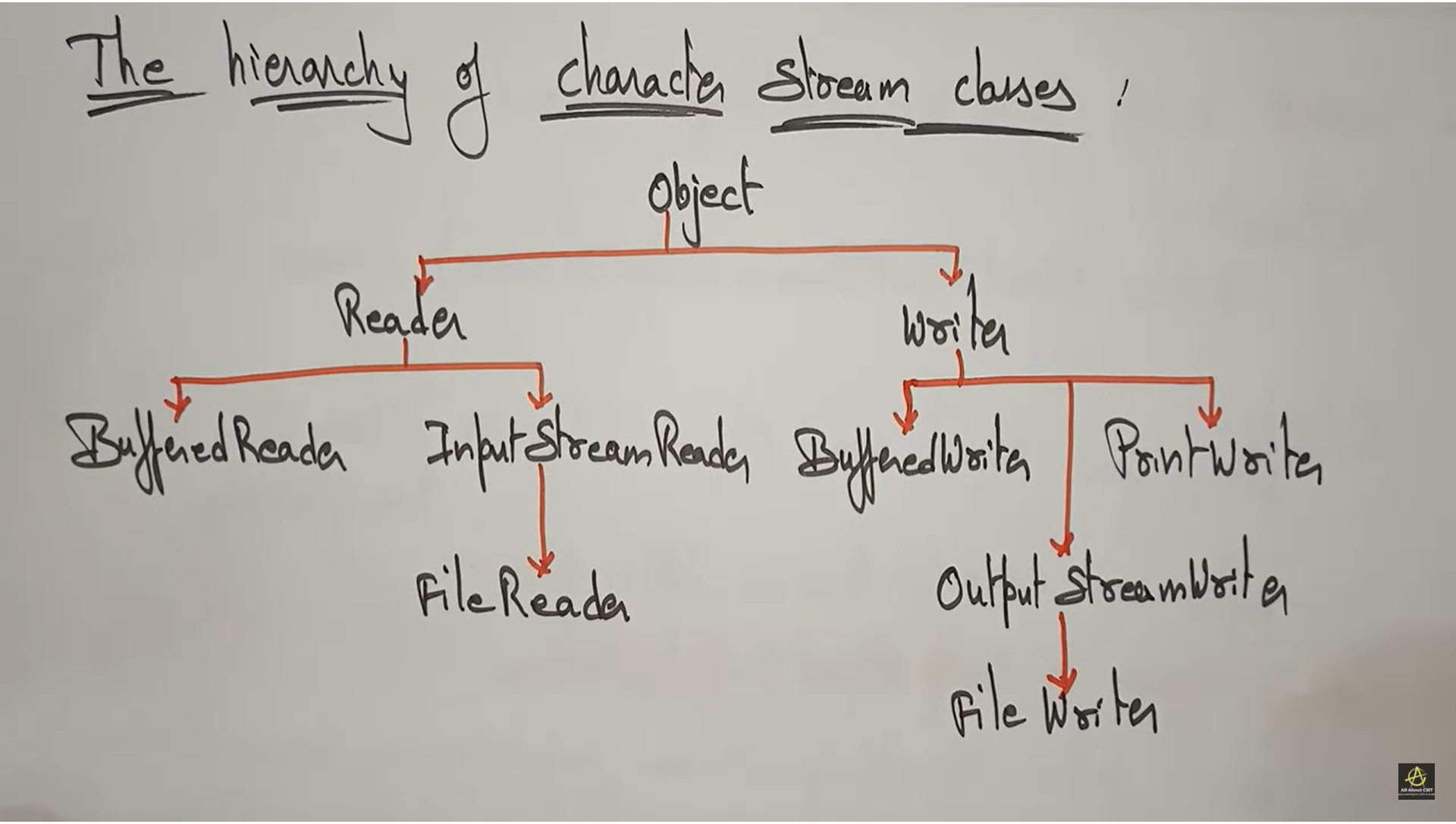
# Input/Output Basics in Java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

## Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter.** Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

# Input/Output Basics in Java



# Input/Output Basics in Java

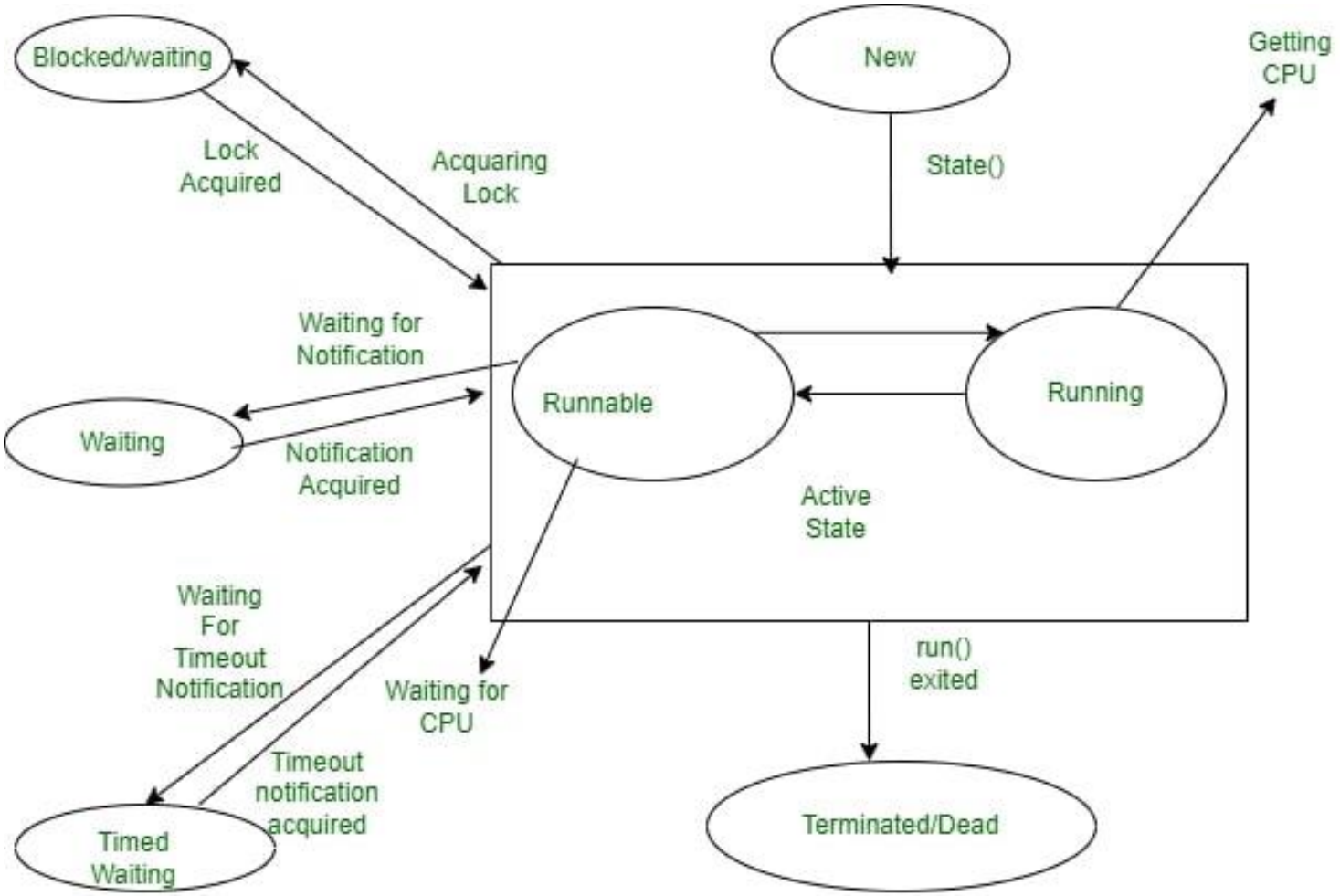
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

# Threads in Java

Threads as a subprocess with lightweight with the smallest unit of processes and also has separate paths of execution. The main advantage of multiple threads is efficiency (allowing multiple things at the same time. For example, in MS Word. one thread automatically formats the document while another thread is taking user input. Another advantage is quick response, if we use multiple threads in a process and if a thread gets stuck due to lack of resources or an exception, the other threads can continue to execution, allowing the process (which represents an application) to continue to be responsive.

# Threads in Java

## Life Cycle Of Thread




Life Cycle Of Thread

# Threads in Java

**1. New State**

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

**2. Active State**

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely:

•**Runnable State:** In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.

•**Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

# Threads in Java

**3. Waiting/Blocked State**

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

# Threads in Java

**4. Timed Waiting State**

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

**5. Terminated State**

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions...etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

# Threads in Java

**Create Threads using Java Programming Language?**

We can create Threads in java using two ways, namely :

1.Extending Thread Class

2.Implementing a Runnable interface

**1. By Extending Thread Class**

We can run Threads in Java by using Thread Class, which provides constructors and methods for creating and performing operations on a Thread, which extends a Thread class that can implement Runnable Interface. We use the following constructors for creating the Thread:

- Thread
- Thread(Runnable r)
- Thread(String name)
- Thread(Runnable r, String name)

## Create Threads by Extending Thread Class

```
import java.io.*;
import java.util.*;

public class GFG extends Thread {
    // initiated run method for Thread
    public void run()
    {
        System.out.println("Thread Started Running...");
    }
    public static void main(String[] args)
    {
        GFG g1 = new GFG();
        // Invoking Thread using start() method
        g1.start();
    }
}
```

# Create Thread by using Runnable Interface

```
import java.io.*;
import java.util.*;
public class GFG implements Runnable {
    // method to start Thread
    public void run()
    {
     System.out.println("Thread is Running Successfully");
    }
    public static void main(String[] args)
    {
        GFG g1 = new GFG();
        // initializing Thread Object
        Thread t1 = new Thread(g1);
        t1.start();
    }
}
```

## Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

### Setter & Getter Method of Thread Priority

The setter and getter method of the thread priority.

**public final int getPriority():**

The java.lang.Thread.getPriority() method returns the priority of the given thread.

**public final void setPriority(int newPriority):**

The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the

# Threads in Java

## 3 constants defined in Thread class:

1.public static int MIN_PRIORITY
2.public static int NORM_PRIORITY
3.public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

# Threads in Java

```
// Importing the required classes
import java.lang.*;

public class ThreadPriorityExample extends Thread
{

// Method 1
// Whenever the start() method is called by a thread  the run() method is inv
oked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}
// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
```

# Threads in Java

```
// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

// 1st Thread  Displaying the priority of the thread  using the getPriority() me
thod
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread  Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread  Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);
```

# Threads in Java

```
// Main thread

// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.current
Thread().getName());

System.out.println("Priority of the main thread is : " + Thread.currentThre
ad().getPriority());

// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThre
ad().getPriority());
}
}
```

# Threads in Java

## Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

# Threads in Java

## Thread Synchronization in Java

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement –

Syntax

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

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

**Thread Synchronization in Java**

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program. There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

# Inter-thread Communication in Java

Inter-thread communication is important when you develop an application where two or more threads exchange some information. Inter-thread communication is achieved by using the wait(), notify(), and notifyAll() methods of the Object class.

# Methods used for Inter-thread Communication

There are three simple methods and a little trick which makes thread communication possible. All the three methods are listed below –

| Sr.No. | Method & Description |
|---|---|
| 1 | **public void wait()**<br>Causes the current thread to wait until another thread invokes the notify(). |
| 2 | **public void notify()**<br>Wakes up a single thread that is waiting on this object's monitor. |
| 3 | **public void notifyAll()**<br>Wakes up all the threads that called wait( ) on the same object. |

These methods have been implemented as **final** methods in Object, so they are available in all the classes. All three methods can be called only from within a **synchronized** context.

# Threads in Java

```
class Chat {
   boolean flag = false;
   public synchronized void Question(String msg) {
      if (flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = true;
      notify();
   }
   public synchronized void Answer(String msg) {
      if (!flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = false;
      notify();
   }
}
```

# Threads in Java

```
class T1 implements Runnable {
   Chat m;
   String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };
   public T1(Chat m1) {
      this.m = m1;
      new Thread(this, "Question").start();
   }
   public void run() {
      for (int i = 0; i < s1.length; i++) {
         m.Question(s1[i]);
      }
   }
}

class T2 implements Runnable {
   Chat m;
   String[] s2 = { "Hi", "I am good, what about you?", "Great!" };
   public T2(Chat m2) {
      this.m = m2;
      new Thread(this, "Answer").start();
   }
   public void run() {
      for (int i = 0; i < s2.length; i++) {
         m.Answer(s2[i]);
      }
   }
}
```

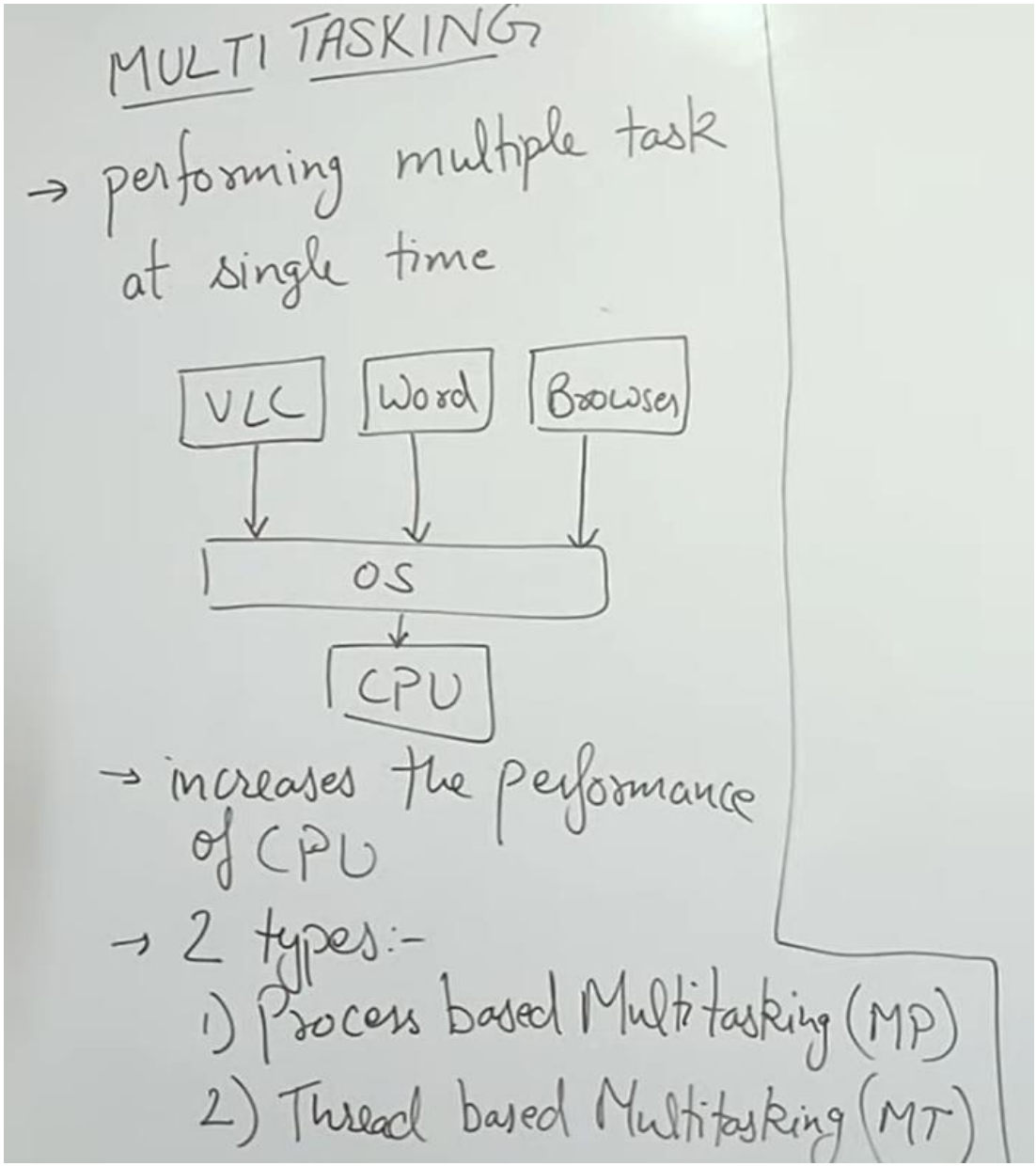
# Threads in Java

```
public class TestThread {
   public static void main(String[] args) {
      Chat m = new Chat();
      new T1(m);
      new T2(m);
   }
}
```

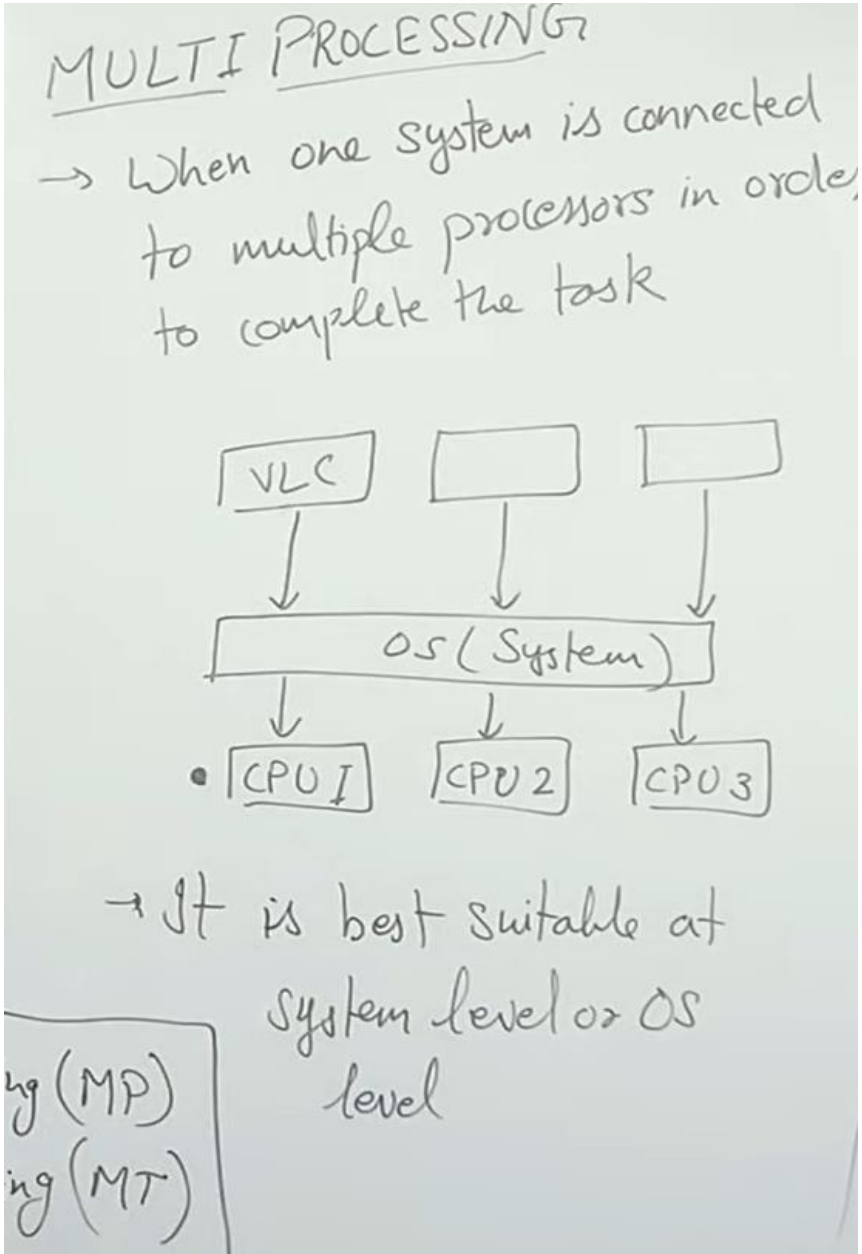
Output

```
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```

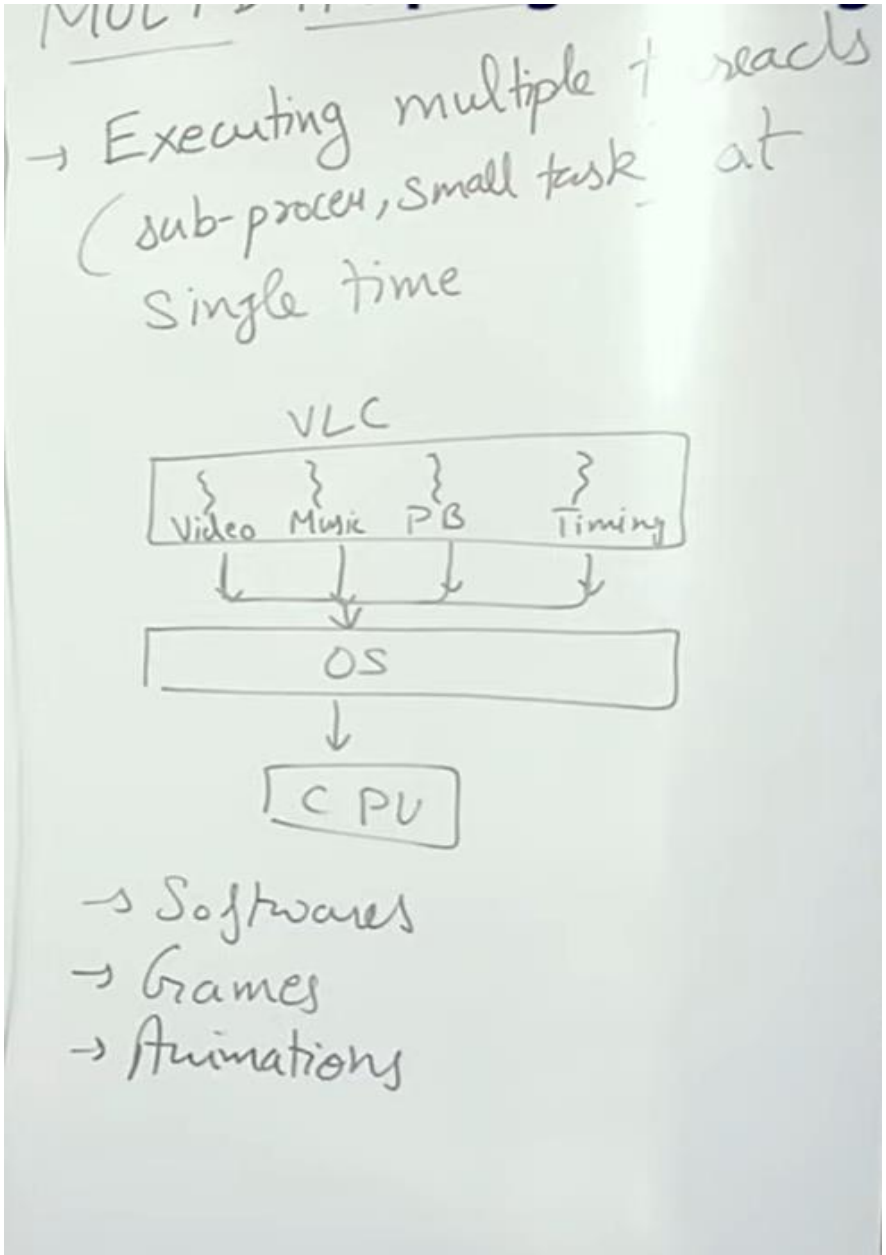
# Threads in Java



# Threads in Java



# Threads in Java



# Threads in Java

→ Multithreading is best suitable at programming level

→ Java provides predefined API for multithreading (10-20%)

- Thread, • Runnable,
- ThreadGroup,
- Concurrency, • Thread Pool

# Threads in Java

## Synchronization Introduction

**1. What is Synchronization ?**

It is the process by which we control the accessability of multiple threads to a particular shared resource

**2. Problem which can occur without synchronization :-**

1. Data Inconsistancy
2. Thread interference

**3. Advantages of Synchronization :-**

1. No data inconsistancy problem
2. No thread interference

**4. Disadvantages of Synchronization :-**

1. Increases the waiting time period of threads
2. Create performance problems

# Threads in Java

## Synchronization Introduction

**1. What is Synchronization ?**

It is the process by which we control the accessability of multiple threads to a particular shared resource

**2. Problem which can occur without synchronization :-**

1. Data Inconsistancy
2. Thread interference

**3. Advantages of Synchronization :-**

1. No data inconsistancy problem
2. No thread interference

**4. Disadvantages of Synchronization :-**

1. Increases the waiting time period of threads
2. Create performance problems

To overcome synchronization disadvantages, java provides one package i.e. java.util.concurrent

# Threads in Java

## How to achieve Synchronization

### Types of Synchronization

2 types of synchronization

**Process Synchronization**

(Not present in java multithreading)

**Thread Synchronization**

2 types of thread synchronization

**Mutual Exclusive**

Can be achieved by 3 ways :-
1. By "Synchronized Method"
2. By "Synchronized Block"
3. By "Static Synchronization"

**Cooperation**

(Inter-thread communication in java)

Can be achieved by following methods of Object class:
1. wait()
2. notify()
3. notifyAll()