Table of Contents

Exceptions Handling in Java

What are exceptions?

An **exception** is any condition that interrupts the normal flow of a program. When an exception occurs, the program terminates unexpectedly and does not continue further unless the exception is handled.

When Can Exceptions Occur?

- **Compile-time** (Checked Exceptions)
- **Runtime** (Unchecked Exceptions)

Examples of Situations that Cause Exceptions:

- Dividing a number by zero
- Accessing an invalid index in an array
- Trying to open a file that doesn't exist

Types of Exceptions in Java

Java exceptions are divided into **three main categories**:

Type	Description	Examples
Checked Exception	Checked at **compile-time**. Must be either handled using try-catch or declared using throws.	ClassNotFoundException, IOException, NoSuchMethodException
Unchecked Exception	Occurs at **runtime**. Not checked by the compiler.	ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException
Errors	Serious problems that are usually **not handled** by programs.	OutOfMemoryError, StackOverflowError

1. What is throw?

✅ **Use:**
throw is used **to actually throw an exception yourself** in the code.

🔧 **Why? (Use cases)**
Sometimes you want to **force an error when something goes wrong**, e.g., if input is invalid.

📌 **Example:**

```
if(age < 0) {
  throw new IllegalArgumentException("Age cannot be negative");
}
```

Explanation:
Here, if age is negative, we **throw** an exception immediately to stop execution and inform the caller about the problem.

2. What is throws?

✅ **Use:**
throws is used in the **method declaration** to say:

"⚠️ This method **might throw this type of exception**, so whoever calls it should handle it."

🔧 **Why? (Use cases)**
To **inform the compiler and programmer** that this method could throw an exception, and it is **not handling it inside itself**.

📌 **Example:**

```
public void readFile(String fileName) throws IOException {
  // code that reads a file
}
```

Explanation:
This says "I am not handling IOException inside this method, so whoever calls readFile() must handle or declare it."

3. What is finally?

✅ **Use:**
finally is a block that **always runs** after try (whether exception happens or not).

🔧 **Why?**
To **clean up resources** like closing files, releasing database connections, etc.

📌 **Example:**

```
try {
  // risky code
}
catch(Exception e) {
  System.out.println("Exception caught");
}
finally {
  System.out.println("I will always run");
}
```

Explanation:
No matter if an exception is thrown or not, **finally block runs**.

Keyword	Use	Nepali Meaning
throw	Actually **throw an exception**	"I am throwing an error now."
throws	**Declares** that a method can throw an exception	"This method might throw this error, caller handle it."
finally	Always runs after try-catch	"For Cleanup ! code here, always runs."

What are Custom Exceptions?

Explanation:

- Java has built-in exceptions like ArithmeticException, IOException, etc.
- Sometimes, you want to create your **own exception class with meaningful names** to handle specific situations in your program.

✅ **How?**

- Create a class extending Exception (checked) or RuntimeException (unchecked).

```
/*
Steps to create a custom exception:
1. Create a new class that extends Exception
2. Create a constructor for the class that takes a message as a parameter
3. Call super(message) in the constructor

And all set you did it

 */
class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg);
    }
}
```

```
/*
    Steps to use the custom exception:
    1. Create a method that throws the custom exception
    2. Call the method in a try block and catch the exception
*/

class Test {  Utsav-56 *
    static int age = 3;  1 usage

    static void setAge(int n) throws InvalidAgeException {
        if (n < 0) {
            throw new InvalidAgeException("Age must be Positive number");
        }
        age = n;
    }

    public static void main(String[] args) {  Utsav-56 *
        try {
            setAge(-4);
        } catch (InvalidAgeException e) {
            e.printStackTrace();
            // or put any message you like
        }
    }
}
```

Explanation:

1. **InvalidAgeException** is a **custom exception class**.
2. **setAge** method **throws** it if the number is negative.
3. main catches and handles it.

2. What are Multiple Catch Blocks?

Explanation:

- You can have **multiple catch blocks** to handle **different types of exceptions** separately.
- The first matching catch block executes.

Program: Multiple Catch Blocks

```
public class MultipleCatchDemo {
  public static void main(String[] args) {
    try {
      String name = null;
      name.length(); // this will throw NullPointerException


      int num = 10 / 0; // this will throw ArithmeticException

    } catch (ArithmeticException e) {
      System.out.println("Arithmetic Exception occurred");
    } catch (NullPointerException e) {
      System.out.println("Null Pointer Exception occurred");
    } catch (Exception e) {
      System.out.println("Some other exception occurred");
    }

  }
}
```

Explanation:

1. **First error:** name.length() causes NullPointerException
2. **first exception is caught** and handled here, so division by zero does not occur because the line below the error line does not execute.

Why is Exception Handling Important?

1. Prevents program from crashing suddenly

Without handling, if an error occurs, the program **stops immediately**.

2. Helps identify and handle errors gracefully

You can **show meaningful messages to the user** instead of confusing errors.

3. Allows the program to continue or shut down properly

You can decide whether to **continue execution or exit safely**, releasing resources.

4. Helps in debugging

You know **where and why** an error happened.

Real life example:
Imagine an ATM machine code. If it crashes due to a small input error without handling, the machine will stop working for everyone. Exception handling ensures **smooth user experience even in errors**.

try-catch-finally Explained

try block:
Contains **code that might cause an exception**.

catch block:
Contains **code to handle that exception** if it occurs.

finally block:
Contains **code that always runs**, whether exception occurs or not (e.g., closing files, releasing database).

Example Program:

```
public class TryCatchFinallyDemo {
  public static void main(String[] args) {
    try {
      int num = 10 / 0; // risky code: division by zero
      System.out.println("Result: " + num);
    }
    catch (ArithmeticException e) {
      System.out.println("Cannot divide by zero.");
    }
    finally {
      System.out.println("This block always runs.");
    }
  }
}
```

Checked vs Unchecked Exceptions

1. Checked Exceptions

Definition:

Exceptions that are **checked at compile time**.

✔ **Compiler requires you to handle them**, either with **try-catch** or by **declaring with throws**.

🔷 **Examples:**

- IOException
- SQLException
- FileNotFoundException

📌 **Why?**

Because they are **expected to happen** during normal program execution (e.g., file not found, network issues) and you should **prepare for them**.

---

2. Unchecked Exceptions

Definition:

Exceptions that are **not checked at compile time**.

✔ **Compiler does NOT force you to handle them.**

🔷 **Examples:**

- ArithmeticException (divide by zero)
- NullPointerException (accessing null object)
- ArrayIndexOutOfBoundsException

📌 **Why?**

These are mostly **programming errors** or **logic mistakes** that can be prevented by writing correct code.

Feature	**Checked Exception**	**Unchecked Exception**
When checked ?	At **compile time**	At **runtime** only
Handling required?	**Yes**, must handle with try catch or declare with throws	**No**, not mandatory
Examples	IOException, SQLException	ArithmeticException, NullPointerException
Type	Subclass of Exception	Subclass of RuntimeException

Throw vs throws

Feature	throw	throws
Purpose	**Actually throws an exception**	**Declares that a method might throw an exception**
Usage	Inside method or block	In method signature
Number of exceptions	Can **throw only one exception at a time**	Can **indicate multiple exceptions**, separated by commas
Example	throw new ArithmeticException("Divide by zero");	public void f() throws IOException, SQLException

What throws Actually Does

In Java, the throws keyword is used in a **method signature** to indicate that the method **might cause** one or more exceptions. It doesn't throw anything itself —

it just **signals** that the method doesn't handle those exceptions internally.

Example:

```
public void readFile(String path) throws IOException {
    // code that might throw IOException
}
```

This tells the compiler and the caller:

"I'm not catching this exception here — you need to handle it." And the caller must handle it gracefully using try-catch

What is Multithreading?

✅ **Definition:**

Multithreading is a **feature that allows multiple parts of a program (threads) to run at the same time** to perform tasks concurrently.

✔️ **Why use it?**

- To **make programs faster** (e.g., downloading files while updating UI).
- To **perform multiple tasks simultaneously** (e.g., audio + video playing together).

🔷 **Real life example:**

Using your phone while downloading apps in the background – **both tasks run together without waiting for each other to finish**.

How is Multithreading implemented in Java?

There are **two main ways**:

1. Extending the Thread class

📌 **Steps:**

1. Create a class that **extends Thread**.
2. Override the **run() method** with the code you want to execute in the thread.
3. Create an object of your class and call **start()** to begin the thread.

Example:

```
class MyThread extends Thread {
  public void run() {
    for(int i=1; i<=5; i++) {
      System.out.println(i + " from thread " + this.getName());
    }
  }

  public static void main(String[] args) {
    MyThread t1 = new MyThread();
    MyThread t2 = new MyThread();
    t1.start(); // starts t1 thread
    t2.start(); // starts t2 thread
  }
}
```

2. Implementing Runnable interface

📌 **Steps:**

1. Create a class that **implements Runnable**.
2. Override the **run() method**.
3. Create a Thread object passing your class object and call **start()**.

🔧 **Example:**

```
class MyRunnable implements Runnable {
  public void run() {
    for(int i=1; i<=5; i++) {
      System.out.println(i + " from " + "MyRunnable");
    }
  }

  public static void main(String[] args) {
    MyRunnable r = new MyRunnable();
    Thread t1 = new Thread(r);
    Thread t2 = new Thread(r);
    t1.start();
    t2.start();
  }
}
```

What is synchronization in multi-threading in java ?

What is Synchronization?

✅ **Definition:**

Synchronization is a **mechanism to control access to shared resources by multiple threads** to prevent **data inconsistency**.

🔧 **Why is it needed?**

When two or more threads access **the same data at the same time**, they might **change its value unexpectedly**, leading to **wrong results (race condition).**

🎯 **Real life example:**

Imagine **two people withdrawing money from the same bank account at the same time** without checking the balance. Both see the same balance and withdraw, leading to **negative balance or wrong final balance**.

✔️ **Synchronization ensures** only one person (thread) accesses it at a time.

How to implement synchronization in Java?

📌 **Using synchronized keyword:**

1. **synchronized method** – only one thread can execute the method at a time.

   Example:

```java
/*

    Adding static before variable and method
    yesto garda hamle instance banai rakhna parena direct
    classname bata call garna pauxa
    sajilo hunxa bhanera static thapeko hai

*/

class Counter {
    static int count = 0;

    static synchronized  void increment() {
        count++;
    }
}


class MyThread extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
            Counter.increment();
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start(); // starts t1 thread
        t2.start(); // starts t2 thread
    }
}
```

What is Inter-Thread Communication?

✅ **Definition:**

Inter-thread communication allows **threads to communicate with each other** (cooperation), rather than working independently.

Why?

To **avoid busy waiting** (when a thread keeps checking a condition repeatedly wasting CPU).

Enables threads to **wait** for each other and **notify** when ready.

Example: Producer-Consumer problem

- **Producer:** produces items and adds to queue.
- **Consumer:** consumes items from queue.
- If queue is full, producer waits. If empty, consumer waits. They **communicate using wait() and notify()**.

Java Methods for Inter-thread Communication

wait()

- Makes the thread **wait (pause) and release lock**, until another thread calls notify() or notifyAll().

notify()

- **Wakes up one waiting thread**.

notifyAll()

- **Wakes up all waiting threads**.

Concept	**Meaning**	**Why?**
Synchronization	Controlling thread access to shared resources	To **avoid data inconsistency**
Inter-thread communication	Threads communicate via wait/notify	To **coordinate work between threads efficiently**

Inheritance:

What is Inheritance?

✅ **Definition:**

Inheritance is a feature of **Object-Oriented Programming (OOP)** where:

A **new class (child/subclass)** can **acquire properties and methods** of an **existing class (parent/superclass)**.

🎯 Why is inheritance useful?

- ✔ **Code Reusability:** No need to write same code again.
- ✔ **Improves readability:** Clear relationship between classes.
- ✔ **Supports hierarchical classification:** Like a real-world family tree.

✅ **Real life example:**

- **Animal (superclass)** → has basic features like eat(), sleep().
- **Dog (subclass)** → inherits eat(), sleep() from Animal, and adds its own bark().

🌟 Features of Inheritance

🔷 **1. Reusability:**
Reuse existing code in the parent class.

🔷 **2. Extensibility:**
Extend features of parent by adding new features in child.

🔷 **3. Data hiding:**
Child class can't directly access private data of parent.

🔷 **4. Overriding:**
Child class can override parent methods to provide specific implementation.

🔷 **5. Hierarchical classification:**
One parent can have multiple child classes.

🌟 **How is Inheritance implemented in Java?**

✅ **Using the extends keyword.**

Example Program

```java
// Parent class
class Animal {  1 usage  1 inheritor
  void sound() {  1 usage
    System.out.println("Animal makes sound....");
  }
}

// Child class
class Dog extends Animal {  2 usages
  void bark() {  1 usage
    System.out.println("Dog barks....");
  }
}

public class InheritanceDemo {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.sound(); // inherited from Animal
    d.bark(); // Dog's own method
  }
}
```

Type	Supported in Java?	Example
Single	✅ Yes	A → B
Multilevel	✅ Yes	A → B → C
Hierarchical	✅ Yes	A → B, A → C
Multiple	❌ No (directly, but interfaces can achieve it)	A, B → C
Hybrid	❌ No (directly)	Combination

🎯 **Final Summary**

- ✔ **Inheritance = acquiring features from another class**
- ✔ Implemented using **extends** keyword
- ✔ Promotes **code reusability, clarity, and OOP structure**

1. Extends keyword

✅ **What is it?**

- extends is used to **create a subclass from a superclass**.
- It establishes **inheritance** between two classes.

🔧 **Where is it used?**

In the **class declaration line** before opening the curly braces

Example code::
Note:: the code for inheritance and Extends is ditttooo same but we will change the name of demo class for good practice from InheritanceDemo to ExtendsDemo, else everything is ditto word to word same

```
// Parent class
class Animal {  1 usage  1 inheritor
  void sound() {  1 usage
    System.out.println("Animal makes sound....");
  }
}

// Child class
class Dog extends Animal {  2 usages
  void bark() {  1 usage
    System.out.println("Dog barks....");
  }
}

public class ExtendsDemo {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.sound(); // inherited from Animal
    d.bark(); // Dog's own method
  }
}
```

2. super keyword

What is it?

- super refers to the **parent class (superclass) object**.
- It is used to:

  a) Access parent class variables and methods (when overridden or hidden in child).

  🔷 **b) Call parent class constructor**.

---

a) Accessing parent method or variable

```
class Animal {  1 usage  1 inheritor
  void sound() {  2 usages  1 override
    System.out.println("Animal makes sound.....");
  }

}

class Dog extends Animal {  2 usages
  @Override  2 usages
  void sound() {
    System.out.println("Dog makes barking sound.....");
  }

  void printEat() {  1 usage
    super.sound(); // calls parents class sound() method
    sound(); // calls Dog's sound()
  }
}

public class SuperDemo {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.printEat();
  }
}
```

Explanation:

- super.sound(); calls **parent class method**.
- Sound(); calls **current class method**.
- We have already overrided the sound so we must use super to call the parent method

b) Calling parent class constructor

```
class Animal {
  Animal() {
    System.out.println("Animal constructor called");
  }
}

class Dog extends Animal {
  Dog() {
    super(); // calls Animal constructor (optional if no-arg)
    System.out.println("Dog constructor called");
  }
}

public class SuperConstructorDemo {
  public static void main(String[] args) {
    Dog d = new Dog();
  }
}
```

✔ **Explanation:**

- super(); calls **Animal's constructor** before executing Dog's constructor body.
-

Final Summary

Keyword	Purpose	Usage
extends	To **inherit** from a parent class	class B extends A { }
super	To **refer to parent class's members** (methods, variables, constructor)	super.methodName() or super();

✔ **extends → creates inheritance relationship**

✔ **super → accesses parent class's members or constructor**

✅ **Inheritance:**
When a **child class acquires properties and methods** of a **parent class** using the extends keyword.

Types of Inheritance

Type	Description	Supported in Java?
Single Inheritance	**One child class inherits from one parent class**	✅ **Yes**
Multilevel Inheritance	**A class inherits from a child class, forming a chain**	✅ **Yes**
Hierarchical Inheritance	**Multiple classes inherit from the same parent class**	✅ **Yes**
Multiple Inheritance	**A class inherits from multiple parent classes**	❌ **No (Java does not support it with classes, only via interfaces)**
Hybrid Inheritance	**Combination of two or more types above**	❌ **No (Java avoids complexity, but interfaces can simulate it)**

1. Single Inheritance

✔ **One child inherits one parent.**

```java
//// Parent class (Super class)
class Animal {  no usages
    void sound() {  no usages
        System.out.println("InheritanceExample.Animal makes a sound");
    }
}


//Child class (Sub class) that inherits from InheritanceExample.Animal
class Dog extends Animal {  no usages

    void bark() {  no usages
        System.out.println("InheritanceExample.Dog barks");
    }
}


// test class to demonstrate single inheritance
public class InheritanceExample {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound(); // Inherited method from Animal class
        myDog.bark();  // Method from Dog class
    }
}
```

2. Multilevel Inheritance

✔ **Chain of inheritance.**

One class extends a class and then the extended class is extended by other.

Lets be clear with following example::

```java
// Base class
class Animal {  no usages
    void eat() {  no usages
        System.out.println("Animal eats food");
    }
}

// Derived class from Animal
class Dog extends Animal {  no usages
    void bark() {  no usages
        System.out.println("Dog barks");
    }
}


// Derived class from Dog
// Multi level inheritance of Animal
class Puppy extends Dog {  no usages
    void weep() {  no usages
        System.out.println("Puppy weeps");
    }
}

class MultilevelInheritance {
    public static void main(String[] args) {

        Puppy myPuppy = new Puppy();

        myPuppy.eat();   // Inherited from Animal
        myPuppy.bark();  // Inherited from Dog
        myPuppy.weep();  // Defined in Puppy
    }
}
```

Explaination:

Dog inherits from animal

Puppy extends from dog

3. Hierarchical Inheritance

✔ **One parent, multiple children.**

```java
// Base class
class Animal {  2 usages  2 inheritors
    void sound() {  2 usages
        System.out.println("Animal makes a sound");
    }
}

// Derived class 1
class Dog extends Animal {  2 usages
    void bark() {  1 usage
        System.out.println("Dog barks");
    }
}

// Derived class 2
class Cat extends Animal {  2 usages
    void meow() {  1 usage
        System.out.println("Cat meows");
    }
}

public class HierarchicalInheritance {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound(); // Inherited from Animal
        myDog.bark();

        Cat myCat = new Cat();
        myCat.sound(); // Inherited from Animal
        myCat.meow();
    }
}
```

Polymorphism

What is Polymorphism?

✅ **Definition:**
Polymorphism is an **Object-Oriented Programming (OOP) concept** where:

The **same function or object behaves differently** in different situations.

Meaning: "One name, many forms."

🎯 **Real life example:**

- A person acts as a **student in class**, **child at home**, and **friend outside**.
  Same person, different behaviors based on context.

Types of Polymorphism in Java

✅ **1. Compile-time polymorphism (Method Overloading)**
✅ **2. Runtime polymorphism (Method Overriding)**

---

1. Compile-time Polymorphism (Method Overloading)

🔷 **What is it?**

When **multiple methods have the same name but different parameters** (different type or number of arguments).

✔ **Decided during compile time**.

How to apply method overloading::

1. Create method with same name
2. Make the argument signature different

Example::

```java
class Calculator{  2 usages
    void add(int a, int b){  1 usage
        System.out.println(a+b);
    }

    void add(int a, int b, int c){  1 usage
        System.out.println(a+b+c);
    }

}

public class OverloadingDemo{

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.add( a: 10,  b: 20);
        calc.add( a: 10,  b: 20,  c: 30);
    }

}
```

Explanation:
Same method name add, **different parameters**.

2. Runtime Polymorphism (Method Overriding)

🔷 **What is it?**

When a **child class provides its own implementation** of a method that is already defined in its parent class.

✔ **Decided during runtime** (which method to call).

Example:

```
class Animal{

    static void sout(String msg){
        System.out.println(msg);
    }

    void sound(){
        sout( msg: "Animal makes sound");
    }

}


class Dog extends Animal{
    @Override
    void sound(){
        sout( msg: "Dog barks");
    }

}


class PolyDemo{
    public static void main(String[] args)
    {
        Animal a = new Animal();
        a.sound();

        Dog d = new Dog();
        d.sound();
    }
}
```

Final Summary::

Type of Polymorphism	How achieved?	When decided?
Compile-time	Method Overloading	Compile time
Runtime	Method Overriding	Runtime

Why is Polymorphism useful?

- ✔ **Increases flexibility and reusability of code.**
- ✔ **Allows one interface to be used for different underlying forms (dynamic behavior).**

Abstraction

the concept of hiding complex implementation details and showing only the essential features of an object is called abstraction

What is an Interface in Java?

✅ **Definition:**
An interface in Java is like a blueprint of a class. It:

- **Contains abstract methods (method declarations without body).**

Purpose:

✔ **To achieve abstraction.**
✔ **To achieve multiple inheritance (since Java does not support multiple inheritance with classes).**
✔ **To define what a class must do, not how.**

Difference between class an Interface

Feature	Class	Interface
Keyword used	class	interface
Members	Can have variables, constructors, concreate methods (with body)	Can have abstract methods (without body)
Implementation	Defines how things work (implementation)	Defines what to do (specification)
Inheritance	Can extend only one class (single inheritance)	It cannot have inheritance
Access Modifiers	Can be public, protected, private	All methods are public abstract by default
Instantiation	Can create objects	Cannot create objects directly (no instantiation)
Constructor	Has constructors	Has no constructor

Memory Tip:

- ✔ **Class = implementation (how)**
- ✔ **Interface = specification (what)**

🎯 **Final Summary**

- ✅ **Interface: Blueprint with method declarations to be implemented by classes.**
- ✅ **Used for achieving abstraction and multiple inheritance in Java.**

What are Abstract Classes?

✅ **Definition:**
An abstract class is a class in Java that:

- **Is declared with the abstract keyword.**
- **Cannot be instantiated directly (cannot create objects of it directly).**
- **Can have both abstract methods (without body) and concrete methods (with body).**

Why use abstract classes?

✔ To provide a common base with partial implementation.
✔ To force child classes to implement specific methods (abstract methods).

How are abstract classes different from regular classes?

Feature	**Abstract Class**	**Regular Class**
Instantiation	Cannot be instantiated	Can be instantiated
Abstract methods	Can have abstract methods	Cannot have abstract methods
Purpose	it provide partial implementation and force child to implement abstract methods	it provide full implementation
Keyword	Declared with abstract keyword	No special keyword needed

Difference between Abstract Class and Interface

Feature	Abstract Class	Interface
Methods	Can have **abstract and concrete methods**	All methods are **abstract by default** (Java 8+: can have default and static methods with body)
Variables	Can have **instance variables**	Can not have instance variables
Inheritance	It Can extend one class	It doesnot support inheritance
Constructors	Can have constructors	**No constructors**
Access Modifiers	Methods can have any access modifier	All methods are **public by default**
Use-case	When classes share common behavior with differences	To define **pure abstraction or contract**

Memory Tip:

✔ **Abstract class = partial abstraction + implementation**
✔ **Interface = full abstraction (contract only)**

🎯 **Final Summary**

✅ **Abstract class:** Cannot be instantiated, used to provide **common base with partial implementation**.
✅ **Interface:** Only defines **what to do (methods to implement)**, no object creation, used for **full abstraction and multiple inheritance**.

Why Multiple Inheritance is not supported directly in Java?

✅ **Multiple inheritance:**
When a class **inherits from more than one parent class**.

Problem:

It creates **ambiguity (diamond problem)** if two parents have methods with the same signature.

Example issue:

If Class C inherits Class A and Class B, and both have show() method, then C.show() – **which one to call?**

Java Solution: Interfaces

✔ Java does not support multiple inheritance with classes.
✔ But it **achieves multiple inheritance using interfaces** because:

- Interfaces only have **method declarations (no implementation by default)**.
- The implementing class provides its **own implementation**, so **no ambiguity**.

How to Implement Multiple Inheritance using Interfaces?

✅ **Steps:**

1. Define **two or more interfaces**.
2. Implement them in a **single class** separated by commas

Example Program

```java
interface Sleepable{
    void sleep();
}

interface Eatable{
    void eat();
}

class Animal implements Sleepable, Eatable{
    @Override
    public void sleep(){
        System.out.println("Sleeping");
    }

    @Override
    public void eat(){
        System.out.println("Eating");
    }
}

public class MultipleInheritanceDemo {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.sleep();
        a.eat();
    }
}
```

Example of abstract class

```java
abstract class Animal {
  abstract void sound(); // abstract method

  void eat() { // concrete method
    System.out.println("Animal eats");
  }
}

class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }

}

public class AbstractDemo {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.sound(); // Dog barks
    d.eat();   // Animal eats
  }
}
```

Explanation:

- **The class animal is declared using abstract keyword (hence it is a abstract class)**
- **It has both concrete and abstract method**
- **Dog is extending the Animal class so it must and must override the abstract method i.e sound**
- **Eat is a concreate method so its implementation is not needed**

File handling

What is File Handling?

✅ **Definition:**
File handling is the process of **creating, reading, writing, and modifying files** using a program.

🔷 **Why is it useful?**

- ✔ To **store data permanently** (unlike variables, which lose data after program ends).
- ✔ To **read or write files** for data processing, reports, logs, etc.

🎯 Real life examples of file handling:

- Saving user data to a .txt file.
- Reading configuration files for app settings.
- Writing logs of transactions in banking systems.

💥 How is File Handling done in Java?

Java provides the **java.io package** for file handling.

☑ Basic File Handling Operations

1. **Creating a File**
2. **Writing to a File**
3. **Reading from a File**
4. **Deleting a File**

Example Programs

Before moving to the examples We should know the following:

1. We must import **java.io.File** && **java.io.IOException and other many file functions** (or just java.io.*) importing * is better as we don't have to remember all those shit functions name
2. Every file operation throws a IOException so we must wrap every file handling operations in a Try catch with IoException
3. To use IoException **java.io.IOException** must be imported (skip if java.io.* is imported)

Creating a new file

```java
import java.io.*;

public class CreateFileExample {

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

            File f = new File( pathname: "example.txt");
            if (f.createNewFile()) {
                System.out.println("File created: " + f.getName());
            } else {
                System.out.println("File already exists.");
            }

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Explanation:

- File class is used to create a instance file.
- createNewFile() creates the file if it does not exist.
- If file already exists it wont do anything and return a False

Writing to a file

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

public class WriteFileExample {

    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter( fileName: "example.txt");
            writer.write( str: "Hello, this is a file writing example.");

            writer.close();
            System.out.println("Successfully wrote to the file.");

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Explanation:

- FileWriter creates a instance of file writer for that file (In java everything is a instance so we need to do so)
- Write() method writes text to files.
- Always call close() to save and close the file.

Reading from a File

```java
import ...

public class ReadFileExample {
    public static void main(String[] args) {

        try {
            File f = new File( pathname: "example.txt");
            Scanner sc = new Scanner(f);

            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                System.out.println(line);
            }

            sc.close();

        } catch (FileNotFoundException e) {
            e.printStackTrace();

        }
    }
}
```

Explanation:

- Scanner class is used to read files line by line.
- We have 2 ways to read but this maybe must simple so this is used

Reading file using FileReader

```
class FileReaderExample {
    public static void main(String[] args) {
        try {

            FileReader fr = new FileReader( fileName: "example.txt");
            BufferedReader reader = new BufferedReader(fr);
            String line;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explaination:

- Both scanner and file reader is used to read file
- Scanner is simple tbh
-

Delete a file

```java
import java.io.File;

public class DeleteFileExample {
    public static void main(String[] args) {

        File f = new File( pathname: "example.txt");

        if (f.delete()) {
            System.out.println("Deleted file: " + f.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }

    }
}
```

Explanation:

- delete() method deletes the file.
- We don't wrap it in try catch because it is not an IO operation we are not performing inout or output so it is IO safe

Operation	**Class/Method Used**
Create	File + createNewFile()
Write	FileWriter + write()
Read	Scanner + nextLine()
Delete	File + delete()

Copying content from one file to another

```java
public class CopyContentExample {
    public static void main(String[] args) {
        try {

            File f = new File( pathname: "example.txt");
            Scanner sc = new Scanner(f);

            File writingFile = new File( pathname: "example2.txt");
            FileWriter fw = new FileWriter(writingFile);

            while (sc.hasNextLine()) {
                String line = sc.nextLine();
                fw.write( str: line+ "\n");
            }


            sc.close();
            fw.close();

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Memory tips:

1. Everything is same as reading file example
2. We only added 2 new lines in line 17 and 18
3. Why we did is we have to also create a file to copy so we need that 2 instances
4. And instead of println we just did fw.write
5. If looked closely everything is ditto same just 2 lines added and even the added line is same as we did in writing file if you remember

Serialization and deserialization

What is Serialization?

✅ **Definition:**

Serialization is the process of **converting an object into a byte stream** so that it can be:

Saved to a file

Transferred over a network

Why we need serialization?

Because files and networks work with **byte data**, not Java objects directly.

What is Deserialization?

Definition:

Deserialization is the process of **converting the byte stream back into an object**.

It **reconstructs the original object** from the saved data.

🎯 **Why are they important?**

✔ **Persistent storage:** We can Save objects to files or databases for later use.

✔ **Communication:** Send objects over networks (e.g., in distributed systems, sockets, RMI).

How to implement Serialization in Java?

Steps:

1. **Implement Serializable interface** in the class. (It is a marker interface,i.e has no methods. So we need to override nothing)
2. Use **ObjectOutputStream class** to write the object.
3. Use **ObjectInputStream** class to read the object.

Memory Tips: (Remember in mind)

Read/ inout = deserialization,
Write/output = serializatiom

- Output for serialization and input for deserialization (for streams)
- We do writeObject to store and readObject to read
- Student class is ditto word by word same no changes

Serialization Example

Before moving to example::

1. **We must import required items**
2. **Here also importing java.io.* is enough you don't need to memorize everything**

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;


class Student implements Serializable {
    int id;
    String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}


class SerializeExample {
    public static void main(String[] args) {
        try {
            Student s1 = new Student( id: 1,  name: "Gwen");

            FileOutputStream fileout = new FileOutputStream( name: "Student.ser");
            ObjectOutputStream objOut = new ObjectOutputStream(fileout);

            objOut.writeObject(s1);

            objOut.close();
            fileout.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explaination:

1. We create object of calss we wanna serialize
2. We make instance of fileOutputStream and objectOutputStream necessary for serializing
3. And then we do objOut.writeObject and pass object we wanna serialize
4. We must close the stream before moving

Deserialization Example:

```java
import java.io.*;

class Student implements Serializable {
    int id;
    String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

class DeserializeEg {

    public static void main(String[] args) {
        try {
            FileInputStream fileIn = new FileInputStream( name: "Student.ser");
            ObjectInputStream objIn = new ObjectInputStream(fileIn);

            Student s = (Student) objIn.readObject();

            objIn.close();
            fileIn.close();

            System.out.println("Success deserializing student object");
            System.out.printf("ID: %d,\t Name: %s%n", s.id, s.name);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explaination::

1. We create instance of FIleInputStream and ObjectInputStream (Necessary)
2. We create the instance of student and parse to Student
3. Optionaally, we are sout the data stored (very good to do)

Memory Tips:

- Output for serialization and input for deserialization (for streams)
- We do writeObject to store and readObject to read
- Student class is ditto word by word same no changes

Streams in Java

What are Streams in Java?

Definition:

Streams are used to **perform input and output operations** in Java by reading/writing data **sequentially**.

Types of Streams

Type	Used for
Byte Stream	**Binary data (images, videos, audio, PDF)**
Character Stream	**Text data (letters, words, sentences)**

Byte Streams

🔷 **What are they?**

- Used to **read and write raw bytes (8-bit data)**.
- Suitable for **binary files**.

🔧 **Classes Used:**

- **InputStream** (abstract class for reading bytes)
  - Example: FileInputStream
- **OutputStream** (abstract class for writing bytes)
  - Example: FileOutputStream

Example: Reading a file using Byte Stream

```java
import java.io.FileInputStream;

public class ByteStreamExample {
    public static void main(String[] args) {

        try {
            FileInputStream f = new FileInputStream( name: "example.txt");

            int i;
            while ((i = f.read()) != -1) {
                char line = (char) i;
                System.out.print(line);
            }

            f.close();

        } catch (Exception e) {
            System.out.println(e);
        }

    }
}
```

Character Streams

What are they?

- Used to **read and write character data (16-bit Unicode characters)**.
- Suitable for **text files**.

Classes Used:

- **Reader** (abstract class for reading characters)
    - Example: FileReader
- **Writer** (abstract class for writing characters)
    - Example: FileWriter

Example: Reading a File using Character Stream

```java
import java.io.FileReader;

public class CharacterStreamExample {
    public static void main(String[] args) {

        try {
            FileReader f = new FileReader( fileName: "example.txt");

            int i;
            while ((i = f.read()) != -1) {
                char line = (char) i;
                System.out.print(line);
            }

            f.close();
        } catch (Exception e) {
            System.out.println(e);
        }

    }
}
```

Memory tips::

1. **Everything is ditto same no change in both byte and character stream**
2. **Only the instance is different else everything is ditttoo word by word same**
3. **SO memorise any one and just change the instance type and all done**

Intro to Strings and String Buffers

What are Strings in Java?

Definition:

A **String** in Java is a **sequence of characters**.
It is an **object of the String class** in java.lang package.

Example:

```
String name = "Hello";
```

Here 'Hello' is called a String literal.

How are Strings handled by JRE/JVM?

1. String Pool (String Constant Pool)

- In Java, strings are stored in a special memory area called **String Pool**.
- When you create: new string
- This will first check the string pool if the string literal already exists in memory
- If not it creates a new memory in pool

And suppose we create

```
String s1 = "Hello";
String s2 = "Hello";
```

Both s1 and s2 **refer to the same string literal** in the String pool. so No new object is created for s2. It will use same memory as s1.

Why store in String Pool?

✔ Saves **memory** (no duplicate strings).
✔ Increases **performance** for string operations.

Why are Strings immutable in Java?

Meaning:

Once a string object is created, **its value cannot be changed**.

Modifying an string by adding or concatenating creates a new string in string pool it don't change the original string so they are immutable

Advantage of string immutability

1. Thread safety:

Immutability makes strings **safe to use in multithreaded environments** without synchronization.

2.Saves **memory** (no duplicate strings).

Final Summary:

Feature	Explanation
String	Sequence of characters, object of String class
Handled by JVM	Stored in **String Pool** for memory efficiency
Immutability	Cannot change value once created; new string object is created for any change
Why immutable?	For **security, memory efficiency, thread safety**

What is StringBuffer in Java?

Definition:

StringBuffer is a **class in Java** used to **create mutable (changeable) strings**.

Unlike String, which is **immutable**, StringBuffer allows you to:

- Modify content without creating new objects.
- Append, insert, delete, and replace characters efficiently.

Why use StringBuffer?

- When you need to **perform many modifications** on strings (e.g., in loops or building dynamic strings), using StringBuffer is **more efficient** than using String.

Difference between Strings and String buffer

Feature	String	StringBuffer
Mutability	**Immutable** (cannot change value once created)	**Mutable** (value can be changed)
Performance	Slower when doing many changes (creates new objects each time)	Faster for multiple modifications (uses same object)
Memory	Creates new object for each modification → more memory used	Uses same object → less memory usage
Thread Safety	Strings are immutable hence thread-safe	StringBuffer is also thread-safe (methods are synchronized)
Use-case	When string content **doesn't change** often	When string content **changes frequently**

Memory Tip:

✔ **String:** Immutable → New object on change.
✔ **StringBuffer:** Mutable → Same object modified.

🎯 **Bonus: StringBuilder**

Similar to StringBuffer we also have stringBuilder but **not synchronized**, hence **faster** but **not thread-safe**. Used when thread safety is **not required**.

Final Summary

- **String:** Immutable, thread-safe, creates new objects for modifications.
- **StringBuffer:** Mutable, synchronized (thread-safe), efficient for many changes.
- **StringBuilder:** Mutable, not synchronized, faster for single-thread use.

String class

String class provides **many useful methods** to perform operations like searching, modifying, comparing, and splitting strings.

Common Methods of String Class

Method	Description
length()	Returns **length of string**
charAt(int index)	Returns **character at given index**
equals(String s)	Checks if **two strings are equal (case-sensitive)**
equalsIgnoreCase(String s)	Checks if **two strings are equal (case-Insensitive)**
contains(CharSequence)	Checks if provided word exists in string
compareTo(String s)	Compares two strings returns 1,0,-1 (same as strComp in c)
startsWith(String)	Checks if string starts with given substring (word)
endsWith(String)	Checks if string ends with given substring (word)
toUpperCase()	Converts to **uppercase**
toLowerCase()	Converts to **lowercase**
trim()	Removes **leading and trailing spaces**
replace(char old, char new)	Replaces old char with new char
indexOf(char)	Returns **index of first occurrence**
lastIndexOf(char)	Returns **index of last occurrence**
split(String regex)	Splits string based on regex
substring(int beginIndex, int endIndex)	Returns **substring from beginIndex to endIndex-1**

Example: Showing String Methods

What is the static keyword in Java?

Definition:

The static keyword, It makes:

Variables,

Methods,

Blocks,

Nested classes

belong to the **class itself rather than to objects**.

Why use static?

- To **share** a common property or method across all objects of the class.
- To **access methods/variables without creating an object**.

Valid items that can be declared as static:

Item	**Explanation**
Static variables (class variables)	Shared by all objects of the class
Static methods	Can be called without object, only access static data
Static blocks	Runs **once** when class is loaded (initialization block)
Static nested classes	Nested classes declared as static

Uses of static keyword

1. **Static Variables (Class level variables)**

   Used when a variable is common to all objects.

2. **Static Methods**

   Can be called without creating object.

   Can access only static data directly.

3. **Static Block**

   Used to initialize static data.

   Executes **once when class is loaded**.

4. **Static Nested Class**

   Nested class declared static can be accessed without outer class object.

What is Iterator in Java?

Definition:

Iterator is an interface in java.util package used to:

- **Iterate over collections (lists, sets, etc.)**
- Access each element **one by one**

🎯 **Key Methods of Iterator:**

Method	**Description**
hasNext()	Returns true if there are more elements
next()	Returns the next element
remove()	Removes current element from collection

What is Comparator in Java?

Definition:

Comparator is an interface in java.util package used to:

- Define **custom sorting order** of objects.
- Compare two objects.

🎯 **Key Method:**

Method	**Description**
compare(T o1, T o2)	Returns **positive, negative, or zero** based on comparison

Diifference between Iterator and Comparator

Feature	**Iterator**	**Comparator**
Purpose	Traverse collection elements	Define custom sorting logic
Package	java.util	java.util
Usage	Iterate using hasNext() and next()	Implement compare() to compare two objects

What is Comparable in Java?

✅ **Definition:**

Comparable is an interface in java.lang package used to:

- ✔ Define **natural ordering** of objects.
- ✔ Allows a class to **compare its objects to each other**.

🎯 **Key Method:**

Method	Description
compareTo(T o)	**Compares current object with specified object and returns: ➔ Negative if current < o ➔ Positive if current > o ➔ Zero if equal**

Feature	Comparable	Comparator
Package	**java.lang**	**java.util**
Method	**compareTo(T o)**	**compare(T o1, T o2)**
Sorting logic defined	**Inside the class itself**	**Outside the class (in separate class)**
Usage	**Defines natural ordering (default sorting)**	**Defines custom ordering (alternative sorting)**
How to use	**Class implements Comparable interface**	**Create separate class implementing Comparator**
Single or multiple sorting	**Only one sorting logic (unless modified)**	**Can define multiple sorting logics with multiple comparators**

Example Summary:

- **Comparable:**
  ➔ Used when a class **has natural default ordering** (e.g., roll number, id).
  ➔ Implement compareTo() method in class.
- **Comparator:**
  ➔ Used when we want to **sort objects differently than natural order** (e.g., by name, age, salary).
  ➔ Implement compare() in a separate class.

Final Tip:

- If you want **default single sorting** → use Comparable.
- If you need **multiple different sorting options** → use Comparator.

Sorting using comparators and Comparables

Using comparables

```
import java.util.*;

class STudent implements Comparable<STudent> {
    int id;
    String name;
    int age;

    STudent(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    // Implementing compareTo method to sort by age
    public int compareTo(STudent s) {
        if (age == s.age)
            return 0;
        else if (age > s.age)
            return 1;
        else
            return -1;
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        ArrayList<STudent> list = new ArrayList<>();
        list.add(new STudent(101, "Ram", 22));
        list.add(new STudent(102, "Shyam", 20));
        list.add(new STudent(103, "Hari", 21));

        Collections.sort(list); // sorts using compareTo()

        System.out.println("Students sorted by age:");
        for (STudent s : list) {
            System.out.println(s.id + " " + s.name + " " + s.age);
        }
    }
}
```

Using comparators

```
import java.util.*;

class Student {
    int id;
    String name;
    int age;

    Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
}

class AgeComparator implements Comparator<STudent> {
    public int compare(STudent s1, STudent s2) {
        if (s1.age == s2.age)
            return 0;
        else if (s1.age > s2.age)
            return 1;
        else
            return -1;
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        ArrayList<STudent> list = new ArrayList<>();

        list.add(new STudent(101, "Ram", 22));
        list.add(new STudent(102, "Shyam", 20));
        list.add(new STudent(103, "Hari", 21));

        System.out.println("Sorting students by age:");
        Collections.sort(list, new AgeComparator());

        for (STudent s : list) {
            System.out.println(s.id + " " + s.name + " " + s.age);
        }
    }
}
```