**What is a Class?**

A **class** in Java is a blueprint or template for creating objects. It defines **fields (class variables)** and **methods (functions)** that describe the behavior and properties of the object.

**Syntax:**

class ClassName {

   // Fields (variables)

   int number;

   String name;

   // Method

   void display() {

      System.out.println("Number: " + number + ", Name: " + name);

   }

}

**What is an Object?**

**An object is an instance of a class.** When a class is defined, no memory is allocated until an object of that class is created **using the new keyword.**

**Syntax:**

ClassName obj = new ClassName(); // obj is created using new keyword

**Example:**

// Define a class

class Car {

   // Fields

   String model;

   int year;

   // Method

   void displayDetails() {

      System.out.println("Model: " + model);

      System.out.println("Year: " + year);

   }

}

// Main class

public class Main {

   public static void main(String[] args) {

       // Create an object

       Car myCar = new Car();

       // Assign values

       myCar.model = "Honda";

       myCar.year = 2025;

       // Call method

      myCar.displayDetails();

   }

}

**Output:**

Model: Honda

Year: 2025

**Key Points:**

| **Term** | **Description** |
| --- | --- |
| **Class** | Template/blueprint for objects |
| **Object** | Instance of a class |
| **Fields** | Variables defined in a class |
| **Methods** | Functions defined in a class |
| **new** | Keyword to create objects |

                                                                          ------------

package pkg\_oops;

class Car{  
  //class variables or instance variables  
  private String model;  
  private int make\_year;  
  
  //Constructor  
  //1. A constructor always has the same name as class name.  
  
  //2. Whenever the object of the class is created or  
  //   instantiated, the default[Car(){}] or user-defined constructor  
  //   of that class is INVOKED and all the class variales  
  //   are initialized to their DEFAULT VALUE.  
  
  Car(String model,int make\_year){  
    //this refers to the object of the current class.  
    this.model=model;  
    this.make\_year=make\_year;  
  }  
  
  //Business Logic  
   public void printDetails() {  
     System.out.println("Model of the Car is: "+model);  
     System.out.println("Year: "+make\_year);  
   }  
}

public class WB7JS {  
  public static void main(String[] args) {  
    //create an object of the class  
    Car car = new Car("Mercedez",2025);  
    car.printDetails();  
  }  
}

Template for the class CAR

-----------------------------

package pkg\_oops;

class Car{  
  //class variables or instance variables  
  private String model;  
  private int make\_year;  
  
  
  
  public Car() {  
    System.out.println("Demo Car for trial run");  
  }

  //Constructor using fields   
  public Car(String model, int make\_year) {  
    this.model = model;  
    this.make\_year = make\_year;  
  }

  //Getters and Setters  
  public String getModel() {  
    return model;  
  }

  public void setModel(String model) {  
    this.model = model;  
  }

  public int getMake\_year() {  
    return make\_year;  
  }

  public void setMake\_year(int make\_year) {  
    this.make\_year = make\_year;  
  }

  @Override  
  public String toString() {  
    return "Car [model=" + model + ", make\_year=" + make\_year + "]";  
  }  
  
}

CarImpl class

---------------

package pkg\_oops;

public class CarImpl {

  public static void main(String[] args) {  
    Car car = new Car("Mercedez",2025);  
    System.out.println(car);  
  
    Car newcar = new Car("BMW",2025);  
    System.out.println(newcar);  
  
    Car democar = new Car();  
  
  }

}

package pkg\_oops;

public class ClassVar {  
  private int id;  
  private String name;  
  
  public void getData() {  
    id=102;  
    name="Steve";  
  }  
  public void printData() {  
    System.out.println("Id: "+id);  
    System.out.println("Name: "+name);  
  }  
  public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ClassVar obj = new ClassVar ();  
    obj.getData();  
    obj.printData();  
  }

}

package pkg\_oops;  
class Student{  
  //instance variables belong to the object  
  int id;  
  String name;  
  int rollNo;  
  
  //static varibales belong to the class  
  // A single copy in the memory and shared by every object.  
  static int groupId;  
}  
public class ClassVar {  
  public static void main(String[] args) {  
    Student s1 = new Student();  
    s1.id=101;  
    System.out.println("Student 1 Id: "+s1.id);  
    Student s2 = new Student();  
    System.out.println("Student 2 Id: "+s2.id);  
  
    Student s3 = new Student();  
    Student s4 = new Student();  
    Student s5 = new Student();  
  
    s3.groupId=101;  
    System.out.println("Group Id for Student 3: "+s3.groupId);  
    System.out.println("Group Id for Student 4: "+s4.groupId);  
    s5.groupId=105;  
    System.out.println("Group Id for Student 3: "+s3.groupId);  
    System.out.println("Group Id for Student 4: "+s4.groupId);  
    System.out.println("Group Id for Student 5: "+s3.groupId);  
  
  
  
  
  }

}

**Variable Types in Java**

In Java, **variables** are containers that hold data values. Each variable must be declared with a **type**, which determines the size and layout of the variable’s memory.

**Types of Variables in Java**

Java variables can be categorized in two main ways:

**1. Based on Data Type:**

**Primitive Types (8 types)**

| **Type** | **Size** | **Example** | **Description** |
| --- | --- | --- | --- |
| byte | 1 byte | byte b = 10; | Small integers (-128 to 127) |
| short | 2 bytes | short s = 1000; | Larger than byte -32768 to +32767 |
| Int | 4 bytes | int x = 25; | Common for integers |
| long | 8 bytes | long l = 123456L; | Very large integers |
| float | 4 bytes | float f = 2.5f; | Decimal numbers (less precision) |
| double | 8 bytes | double d = 3.1415; | Decimal numbers (more precision) |
| char | 2 bytes | char c = 'A'; | Single characters |
| boolean | 1 bit | boolean b = true; | True or false |

**Reference or Non-Primitive  Types**

* Store references (addresses) to objects.
* Examples:

String name = "Alice"; // String is a reference type

int[] arr = {1, 2, 3}; // Arrays are reference types

Car myCar = new Car(); // Custom class object

**2. Based on Scope and Lifetime:**

**Local Variables**

* Declared inside methods or blocks.
* Only accessible within the method.
* Must be initialized before use.

void print() {

   int x = 10; // local variable

  System.out.println(x);

}

**Instance Variables (Non-static Fields)**

* Declared inside a class but **outside** methods.
* Each object gets its own copy.

class Dog {

   String breed; // instance variable

}

**Static Variables (Class Variables)**

* Declared with the static keyword.
* Shared among all instances of the class.

class Counter {

   static int count = 0; // static variable

}

**Summary Table:**

| **Variable Type** | **Scope** | **Stored in** | **Example** |
| --- | --- | --- | --- |
| Local Variable | Method/block | Stack | int x = 5; in a method |
| Instance Variable | Object | Heap | String name; in a class |
| Static Variable | Class | Method Area | static int count = 0; |

Person Class

---------------

package pkg\_oops;

// Implementing one of the OOPS principles - Encapsulation  
public class Person {  
  // Private variables - data hiding  
  private String name;  
  private int age;

  // Public getter method for name  
  public String getName() {  
    return name;  
  }

  // Public setter method for name  
  public void setName(String newName) {  
    name = newName;  
  }

  // Public getter method for age  
  public int getAge() {  
    return age;  
  }

  // Public setter method for age  
  public void setAge(int newAge) {  
    if (newAge > 0) {  
      age = newAge;  
    } else {  
      System.out.println("Age must be positive.");  
    }  
  }  
}

Person Implementation

--------------------------

package pkg\_oops;

public class PersonImpl {

  public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    Person person = new Person();  
  
    //Setting the variable values  
    person.setName("Alex");  
    person.setAge(21);  
  
    //Getting the Person details  
    System.out.println("Person Name: "+person.getName());  
    System.out.println("Person Age: "+person.getAge());  
  }

 }

**Encapsulation in Java**

**Definition:**

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) principles. It refers to the practice of **wrapping data (variables)** and **code (methods)** together into a single unit (class), and **restricting direct access** to some of the object's components. This is usually done by:

1. Making variables **private**.
2. Providing **public getter and setter methods** to access and update the values.

This helps protect the internal state of an object from unintended or harmful modifications and promotes modular, maintainable code.

**Example in Java:**

// A simple example of encapsulation

public class Person {

   // Private variables - data hiding

   private String name;

   private int age;

   // Public getter method for name

   public String getName() {

       return name;

   }

   // Public setter method for name

   public void setName(String newName) {

       name = newName;

   }

   // Public getter method for age

   public int getAge() {

       return age;

   }

   // Public setter method for age

   public void setAge(int newAge) {

       if (newAge > 0) {

           age = newAge;

       } else {

          System.out.println("Age must be positive.");

       }

   }

}

**Implementation:**

public class Main {

   public static void main(String[] args) {

       Person p = new Person();

      p.setName("Alice");

       p.setAge(25);

**Abstraction in Java:**

**Definition:**

**Abstraction** is an Object-Oriented Programming (OOP) principle that **hides complex internal implementation** and **shows only the essential features** of an object.

In Java, **abstraction is achieved using:**

1. **Abstract classes**
2. **Interfaces**

This allows the user to focus on **what an object does**, instead of **how it does it**.

**Abstraction Example using java.lang.Number**

java.lang.Number is an **abstract class** in Java. It serves as a **base class for numeric wrapper classes** like Integer, Double, Float, etc.

**Hierarchy Overview:**

java.lang.Number (abstract)

│

├── Integer

├── Double

├── Float

├── Long

└── Short

**Here's how it works:**

* Number defines abstract methods like intValue(), doubleValue(), etc.
* Subclasses like Integer and Double implement those methods.

**Example:**

public class AbstractionExample{

   public static void main(String[] args) {

       // Using the abstract class Number via its subclass Integer

       Number num = new Integer(42);

      System.out.println("Integer value: " + num.intValue());

      System.out.println("Double value: " + num.doubleValue());

   }

}

**Explanation:**

* Number is abstract — you can't create an instance of it directly.
* Integer is a concrete subclass of Number.
* The intValue() and doubleValue() methods are **abstract methods in Number**, implemented by Integer.

**Conclusion:**

This is a real-world abstraction example in the Java core API:

* The Number class **hides the implementation details**.
* Concrete subclasses like Integer and Double provide **specific behaviors**.
* //CreditCardPayment  
  class CreditCardPayment {  
    public void makePayment(double amount) {  
      amount = amount-(0.2\*amount);  
      System.out.println("Credit card payment of $" + amount + " processed.");  
    }  
  }  
    
  //PayPalPayment  
  class PayPalPayment {  
    public void makePayment(double amount) {  
      amount = amount-100;  
      System.out.println("PayPal payment of $" + amount + " processed.");  
    }  
  }  
    
  class CreditCard {  
    public static void main(String[] args) {  
      CreditCardPayment creditPay = new CreditCardPayment();  
      creditPay.makePayment(1000.0);
* PayPalPayment paypal = new PayPalPayment();  
      paypal.makePayment(2000.0);
* }  
  }
* **What is Inheritance in Java?**
* **Inheritance** is a feature in Java where a class (called **subclass** or **child**) inherits fields and methods from another class (called **superclass** or **parent**).  
  It promotes **code reusability** and establishes **a parent-child relationship** between classes.
* **Syntax:**
* class Child extends Parent {
* // Child class inherits Parent's properties and methods
* }
* **Types of Inheritance in Java**
* **1. Single Inheritance**
* **One class inherits from one superclass.**
* class Animal {
* void sound() {
* System.out.println("Animal makes a sound");
* }
* }
* class Dog extends Animal {
* void bark() {
* System.out.println("Dog barks");
* }
* }
* public class Main {
* public static void main(String[] args) {
* Dog d = new Dog();
* d.sound(); // Inherited method
* d.bark();
* }
* }
* **2. Multilevel Inheritance**
* **A class inherits from another class, which itself inherits from another.**
* class Grandparent {
* void show1() {
* System.out.println("Grandparent method");
* }
* }
* class Parent extends Grandparent {
* void show2() {
* System.out.println("Parent method");
* }
* }
* class Child extends Parent {
* void show3() {
* System.out.println("Child method");
* }
* }
* public class Main {
* public static void main(String[] args) {
* Child c = new Child();
* c.show1(); // from Grandparent
* c.show2(); // from Parent
* c.show3(); // from Child
* }
* }
* **3. Hierarchical Inheritance**
* **Multiple classes inherit from a single parent class.**
* class Vehicle {
* void start() {
* System.out.println("Vehicle started");
* }
* }
* class Car extends Vehicle {
* void drive() {
* System.out.println("Car is driving");
* }
* }
* class Bike extends Vehicle {
* void ride() {
* System.out.println("Bike is riding");
* }
* }
* public class Main {
* public static void main(String[] args) {
* Car car = new Car();
* Bike bike = new Bike();
* car.start();
* car.drive();
* bike.start();
* bike.ride();
* }
* }
* **4. Multiple Inheritance (using Interfaces)**
* Java doesn't support multiple inheritance with classes, but you can achieve it using **interfaces**.

class MathUtils {  
    // Method with 1 int parameter  
    int add(int a) {  
        return a;  
    }

    // Method with 2 int parameters  
    int add(int a, int b) {  
        return a + b;  
    }

    // Method with 3 int parameters  
    int add(int a, int b, int c) {  
        return a + b + c;  
    }  
  double add(double a,int b,int c){  
    return a\*b\*c;  
  }  
}

class Main {  
    public static void main(String[] args) {  
        MathUtils mu = new MathUtils();  
        System.out.println(mu.add(5));            // Output: 5  
        System.out.println(mu.add(5, 10));        // Output: 15  
        System.out.println(mu.add(5, 10, 15));    // Output: 30  
    System.out.println(mu.add(5.0, 10, 15));    // Output: 30  
  
  }  
}

class Animal {  
    void sound() {  
        System.out.println("Animal makes a sound");  
    }  
}

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

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

class Main {  
    public static void main(String[] args) {  
        Animal a; // reference of superclass  
    a = new Dog();  
        a.sound(); // Output: Dog barks

        a = new Cat();  
        a.sound(); // Output: Cat meows  
    }  
}

**Polymorphism**

**Polymorphism** in Java is one of the core concepts of object-oriented programming (OOP). It allows objects of different classes to be treated as objects of a common super class. The main goal of polymorphism is to write more flexible and reusable code.

**Types of Polymorphism in Java**

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

**1. Compile-Time Polymorphism (Method Overloading)**

This occurs when multiple methods have the same name but different parameter lists in the same class.

**Example:**

class MathUtils {

   // Method with 1 int parameter

   int add(int a) {

       return a + 0;

   }

   // Method with 2 int parameters

   int add(int a, int b) {

       return a + b;

   }

   // Method with 3 int parameters

   int add(int a, int b, int c) {

       return a + b + c;

   }

}

public class Main {

   public static void main(String[] args) {

       MathUtils mu = new MathUtils();

      System.out.println(mu.add(5));           // Output: 5

      System.out.println(mu.add(5, 10));       // Output: 15

      System.out.println(mu.add(5, 10, 15));   // Output: 30

   }

}

**2. Runtime Polymorphism (Method Overriding)**

This occurs when a subclass provides a specific implementation of a method already defined in its superclass.

**Example:**

class Animal {

   void sound() {

      System.out.println("Animal makes a sound");

   }

}

class Dog extends Animal {

   @Override

   void sound() {

      System.out.println("Dog barks");

   }

}

class Cat extends Animal {

   @Override

   void sound() {

      System.out.println("Cat meows");

   }

}

public class Main {

   public static void main(String[] args) {

       Animal a; // reference of superclass

a = new Dog();

       a.sound(); // Output: Dog barks

       a = new Cat();

       a.sound(); // Output: Cat meows

   }

}

**Key Benefits of Polymorphism**

* Code reusability
* Loose coupling
* Easy to scale and maintain

**Abstract class**

An **abstract class** is a class that cannot be instantiated on its own and may contain **abstract methods** (methods without a body) as well as **concrete methods** (methods with a body). It serves as a **base class** that other classes can extend to provide implementations for the abstract methods.

**Syntax of an Abstract Class**

abstract class Animal {

   // Abstract method (no body)

   abstract void sound();

   // Concrete method

   void eat() {

      System.out.println("This animal eats food.");

   }

}

**Example: Abstract Class**

// Abstract class

abstract class Animal {

   abstract void sound();  // Abstract method

   void sleep() {  // Concrete method

      System.out.println("Sleeping...");

   }

}

// Subclass that provides implementation

class Dog extends Animal {

   void sound() {

      System.out.println("Dog barks");

   }

}

public class Main {

   public static void main(String[] args) {

       Animal myDog = new Dog();

       myDog.sound();  // Outputs: Dog barks

      myDog.sleep();  // Outputs: Sleeping...

   }

}

**Key Points**

* An abstract class **cannot be instantiated** directly.
* It can have both **abstract and non-abstract methods**.
* A class that extends the abstract class must **implement all its abstract methods**, unless it's also declared abstract.
* Abstract classes can have **constructors**, **fields**, and **static methods**.

**When to Use Abstract Classes**

Use abstract classes when:

* You want to define a common base class with default behavior.
* You expect subclasses to implement specific methods.
* You want to enforce a contract but also provide partial implementation.

package pkg\_oops;  
//Abstract class  
//We cannot instantiate the object of an abstract class.  
abstract class Animal {  
  
  //Constructor  
  Animal(){  
    System.out.println("Abstract class Constructor");  
  }  
  
  //abstract method  
  abstract void sound();  // Abstract method

//concrete method  
  void sleep() {  // Concrete method  
     System.out.println("Sleeping...");  
}  
  //Fields  
  static int abs\_var;  
  
}

//Subclass that provides implementation  
class Dog extends Animal {  
Dog(){  
   //This has to be the 1st statement in child class Constructor.  
   super();  
   System.out.println("Dog class Constructor...");  
}  
  
  void sleep() {  
    super.sleep();  
    System.out.println("Dog is sleeping");  
  }  
  
  void sound() {  
     System.out.println("Dog barks");  
}  
}

class AbstractClassEx {  
public static void main(String[] args) {  
     Dog myDog = new Dog();  
     myDog.sound();  // Outputs: Dog barks  
     myDog.sleep();  // Outputs: Sleeping...  
     myDog.abs\_var=1001;  
     System.out.println("Variable in an abstract class: "+myDog.abs\_var);  
}  
}

package pkg\_oops;  
//Abstract class  
//We cannot instantiate the object of an abstract class.  
abstract class Animal {  
  
  //Constructor  
  Animal(){  
    System.out.println("Abstract class Constructor");  
  }  
  
  //abstract method  
  abstract void sound();  // Abstract method

//concrete method  
  void sleep() {  // Concrete method  
     System.out.println("Sleeping...");  
}  
  //Fields  
  static int abs\_var;  
  int non\_static\_field;  
  
  //static methods  
  static String animalStay(String loc) {  
    return loc;  
  }  
}

//Subclass that provides implementation  
class Dog extends Animal {  
Dog(){  
   //This has to be the 1st statement in child class Constructor.  
   super();  
   System.out.println("Dog class Constructor...");  
}  
  
  void sleep() {  
    super.sleep();  
    System.out.println("Dog is sleeping");  
  }  
  
  void sound() {  
     System.out.println("Dog barks");  
}  
}

class AbstractClassEx {  
public static void main(String[] args) {  
     Dog myDog = new Dog();  
     myDog.sound();  // Outputs: Dog barks  
     myDog.sleep();  // Outputs: Sleeping...  
     Dog.abs\_var=1001;  
     System.out.println("Variable in an abstract class: "+Dog.abs\_var);  
     System.out.println(Dog.animalStay("Farm House"));  
}  
}

**Interface in Java**

An **interface** is a reference type that defines a **contract** for what a class can do, without specifying how it does it. It is a **blueprint of a class**, containing only **abstract methods** (until Java 7), and from Java 8 onwards, it can also contain **default** and **static methods**.

**Syntax of an Interface**

interface Car{

   void sound(); // abstract method

}

**Example: Interface in Action**

// Define an interface

interface Animal {

   void sound();  // abstract method

}

// Implementing the interface

class Dog implements Animal {

   public void sound() {

      System.out.println("Dog barks");

   }

}

class Cat implements Animal {

   public void sound() {

      System.out.println("Cat meows");

   }

}

public class Main {

   public static void main(String[] args) {

       Animal myDog = new Dog();

       Animal myCat = new Cat();

      myDog.sound();  // Outputs: Dog barks

      myCat.sound();  // Outputs: Cat meows

   }

}

**Key Features of Interfaces**

| **Feature** | **Description** |
| --- | --- |
| **Methods** | Abstract by default (until Java 7). Java 8+ allows default and static methods. |
| **Fields** | public, static, and final by default (constants). |
| **Inheritance** | A class can implement **multiple interfaces** (unlike extending only one class). |
| **Access Modifiers** | All interface methods are implicitly public abstract. |

package pkg\_oops;

public interface ICar {  
  public abstract void gears();  
  void engine();  
  void color();  
  void frontMirror();  
  void rearMirror();  
  void brakeSystem();  
  void acceleration();  
  void tyreDia();  
  void fuelTank();  
  void airBags();  
  void soundSystem();  
  void steeringType();  
  void maxSpeed();  
  void mileage();  
  void bootSpace();  
  void seatingCapacity();  
}  
interface ICar2 extends ICar{  
  void autoParking();  
}

package pkg\_oops;

//class Mercedez implements ICar,ICar2{  
class Mercedez implements ICar2{

  public void gears() {  
    System.out.println("7 gears");  
  }  
  public void steeringType() {  
    System.out.println("With Sound Controls and Cruise Control");  
  }  
  public void tyreDia() {  
    System.out.println("90 cms");  
  }  
  public void engine() {  
    System.out.println("1600 cc");  
  }  
  public void mileage() {  
    System.out.println("12 kms");  
  }  
  public void acceleration() {  
    System.out.println("0 to 100 in 10 secs");  
  }  
  public void seatingCapacity() {  
    System.out.println("7 seater");  
  }  
  public void color() {  
    System.out.println("Black");  
  }  
  public void bootSpace() {  
    System.out.println("450 Ltr.");  
  }  
  public void frontMirror() {  
    System.out.println("2 front mirrors");  
  }  
  public void rearMirror() {  
    System.out.println("4 rear mirrors");  
  }  
  public void fuelTank() {  
    System.out.println("60 litres");  
  }  
  public void soundSystem() {  
    System.out.println("JBL Sound System");  
  }  
  public void airBags() {  
    System.out.println("8 airbags");  
  }  
  public void maxSpeed() {  
    System.out.println("Maxspeed: 350");  
  }  
  public void brakeSystem() {  
    System.out.println("Dual ATS brake System");  
  }  
  
  @Override  
  public void autoParking() {  
    System.out.println("AutoParking Fetaure is present in Mercedez G9");  
  }  
}

package pkg\_oops;

public class CarReady {

  public static void main(String[] args) {  
    ICar2 mercedez = new Mercedez();  
    mercedez.acceleration();  
    mercedez.airBags();  
    mercedez.bootSpace();  
    mercedez.brakeSystem();  
    mercedez.color();  
    mercedez.engine();  
    mercedez.frontMirror();  
    mercedez.fuelTank();  
    mercedez.gears();  
    mercedez.maxSpeed();  
    mercedez.mileage();  
    mercedez.rearMirror();  
    mercedez.seatingCapacity();  
    mercedez.soundSystem();  
    mercedez.steeringType();  
    mercedez.tyreDia();  
    mercedez.autoParking();  
  
  }

}

**Java 8 interface enhancements**

Java 8 introduced **significant enhancements** to interfaces, allowing them to include behavior (code), not just declarations. This was a major shift from previous versions of Java.

**Java 8 Interface Enhancements**

1. **Default Methods**
2. **Static Methods**
3. **Functional Interfaces –**This interface has only 1 abstract method and if you annotate the interface with @FunctionalInterface then you cannot ADD ANOTHER ABSTRACT METHOD.
4. **@FunctionalInterface Annotation**

**1. Default Methods**

* Allow interfaces to have **concrete methods** with default implementation.
* Useful for adding methods to existing interfaces **without breaking implementing classes.**

interface MyInterface {

   default void show() {

      System.out.println("Default show implementation");

   }

}

**Implementing class can use or override this method.**

class MyClass implements MyInterface {

   // Optional override

}

**2. Static Methods in Interfaces**

* Interfaces can now have static utility methods.

interface MyInterface {

   static void display() {

      System.out.println("Static method in interface");

   }

}

Called using the interface name:

MyInterface.display();

**3. Functional Interfaces**

* An interface with **only one abstract method** (can have multiple default/static methods).
* Used heavily in **lambda expressions** and **Streams API**.

@FunctionalInterface

interface Calculator {

   int operate(int a, int b);

}

**Example with lambda:**

Calculator add = (a, b) -> a + b;

System.out.println(add.operate(5, 3)); // Output: 8

**4. @FunctionalInterface Annotation**

* Optional but helpful annotation.
* Ensures the interface has **exactly one abstract method**.
* Compiler will throw an error if more than one is added.

@FunctionalInterface

interface Converter {

   String convert(String input);

}

**Summary Table**

| **Feature** | **Description** |
| --- | --- |
| default methods | Concrete methods with default logic |
| static methods | Utility methods called via interface name |
| Functional interfaces | One abstract method (used with lambdas) |
| @FunctionalInterface | Compile-time validation for functional interfaces |

**Lambda Expression**

Lambda expressions in Java were introduced in **Java 8** as part of the effort to support **functional programming**. They provide a clear and concise way to represent **one method interface** (i.e., **functional interfaces**) using an **anonymous function**—meaning a function without a name.

**What Is a Lambda Expression?**

A **lambda expression** is a short block of code that takes in parameters and returns a value. It can be passed around and executed later.

**Basic Syntax:**

()-> expression

Or

(parameters) -> expression

Or:

(parameters) -> {statements;}

**Lambda Expression Syntax**

| **Component** | **Example** | **Description** |
| --- | --- | --- |
| No parameters | () -> System.out.println("Hi") | No input arguments |
| One parameter | x -> x \* x | Parentheses optional for one arg |
| Multiple params | (x, y) -> x + y | Must use parentheses |
| With body | (x, y) -> { return x + y; } | Use {} when multiple statements |

**Example Using Custom Functional Interface**

// Step 1: Create a functional interface

@FunctionalInterface

interface Greet {

   void sayHello();   // No parameters

}

public class LambdaNoParam {

   public static void main(String[] args) {

       // Step 2: Use lambda expression to implement the interface

       Greet greet = () -> System.out.println("Hello from lambda!");

       // Step 3: Call the method

      greet.sayHello();

   }

}

**Example 2**

**Custom Functional Interface with parameters**

@FunctionalInterface

interface MathOperation {

   int operate(int a, int b);

}

public class LambdaExample {

   public static void main(String[] args) {

       MathOperation add = (a, b) -> a + b;

      System.out.println(add.operate(5, 3));

   }

}

**Example with Body (Custom Functional Interface)**

@FunctionalInterface

interface Greet {

   void sayHello();

}

public class LambdaExample {

   public static void main(String[] args) {

       // Lambda with a multi-line body

       Greet greet = () -> {

          System.out.println("Hello!");

          System.out.println("This is a multi-line lambda body.");

       };

      greet.sayHello();

   }

}

**Example: Add, Subtract, Multiply, Divide Using Lambdas**

**// Step 1: Create a functional interface**

@FunctionalInterface

interface MathOperation {

   double operate(double a, double b);

}

class Main{

   public static void main(String[] args) {

**// Step 2: Define lambda expressions for each operation**

       MathOperation add = (a, b) -> a + b;

       MathOperation subtract = (a, b) -> a - b;

       MathOperation multiply = (a, b) -> a \* b;

       MathOperation divide = (a, b) -> {

           if (b == 0) {

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

               return 0;

           }

           return a / b;

       };

**// Step 3: Use the lambdas**

       double x = 100, y = 50;

      System.out.println("Addition: " + add.operate(x, y));

      System.out.println("Subtraction: " + subtract.operate(x, y));

      System.out.println("Multiplication: " + multiply.operate(x, y));

      System.out.println("Division: " + divide.operate(x, y));

   }

}

**Example with return statement**

@FunctionalInterface

interface MathOperation {

   double operate(double a, double b);

}

public class LambdaExample {

   public static void main(String[] args) {

       // Lambda expressions with return statements

       MathOperation add = (a, b) -> { return a + b; };

       MathOperation subtract = (a, b) -> { return a - b; };

       MathOperation multiply = (a, b) -> { return a \* b; };

       MathOperation divide = (a, b) -> {

           if (b == 0) {

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

               return 0;

           }

           return a / b;

       };

       double x = 100, y = 40;

      System.out.println("Addition: " + add.operate(x, y));

      System.out.println("Subtraction: " + subtract.operate(x, y));

      System.out.println("Multiplication: " + multiply.operate(x, y));

      System.out.println("Division: " + divide.operate(x, y));

   }

}

**Packages**

In Java, **packages** are used to group related classes, interfaces, and sub-packages together. They help:

* **Organize code** logically
* **Avoid class name conflicts**
* **Control access** using access modifiers
* Make code **easier to maintain and reuse**

**Types of Packages in Java**

1. **Built-in Packages**
   * Provided by Java API
   * Examples:
     + java.lang – core classes (String, Math, etc.)
     + java.util – data structures (ArrayList, HashMap)
     + java.io – input/output classes
     + java.net – networking
     + javax.swing – GUI components
     + java.sql - JDBC
2. **User-defined Packages**
   * Created by the programmer to organize their own classes

**Creating a Package (User-defined)**

1. Create a file called MyClass.java:

package mypackage;  // Declare package at the top

public class MyClass {

   public void display() {

      System.out.println("Hello from MyClass in mypackage!");

   }

}

1. Compile with:

javac -d . MyClass.java

This creates a folder named mypackage with the compiled class inside.

**Using a Package**

import mypackage.MyClass;

public class Test {

   public static void main(String[] args) {

       MyClass obj = new MyClass();

       obj.display();

   }

}

Compile both classes, then run the Test class.

**Important Points**

* package declaration must be the first line (excluding comments).
* Use import to access classes from other packages.
* The -d flag in javac sets the destination for class files, helping maintain package structure.

Practice----------------

package pack1;

public class Pack1\_Class {  
  public int pack1\_int;  
  protected double pack1\_doub;  
  private String pack1\_name;  
  
  public void printData(int a,int b) {  
  
    System.out.println("Addition is: "+(a+b));  
  }  
  
}

package pkg\_oops;  
import pack1.Pack1\_Class;  
public class Class\_2 {

  public static void main(String[] args) {  
    Pack1\_Class obj = new Pack1\_Class();  
    obj.printData(200, 100);  
    obj.pack1\_int=1234;  
    System.out.println(obj.pack1\_int);  
  
  }

}

/\*Inner class ---> A class defined in another class:  
1.  Outer class  
2. Inner class.

Why use Inner classes:  
----------------------  
1.  To make code cleaner.  
2.  Private methods or instance variables in an outer class  
  can be accessed usin g inner class.  
\*/  
class OuterClass{  
  private String str="Hello Learners";  
  
  class InnerClass{  
    public void printText(){  
      System.out.println("Outer class Private Variable: "+str);  
    }  
  }  
  
  public static void main(String[] args){  
    OuterClass outer = new OuterClass();  
    OuterClass.InnerClass inner= outer.new InnerClass();  
    //It is because the innerclass holds an implicit  
    // reference to the outer class instance;  
    inner.printText();  
  }  
}

//Anonymous inner class

interface Emp{  
  int empid=101;  
  void getEmpId();  
}

class Employee{  
    public static void main(String[] args){  
      Emp emp = new Emp(){  
        public void getEmpId(){  
          System.out.println("EmpId of an employee is: "+  empid);  
        }  
      };  
      emp.getEmpId();  
    }  
}

**Exception Handling**

Exception handling in Java is a powerful mechanism that allows developers to manage runtime errors, **maintain normal application flow,** and improve program reliability and user experience.

**What is an Exception?**

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions. It typically occurs due to:

* Invalid user input
* File not found
* Network issues
* Division by zero
* Accessing an invalid array index, etc.

**Types of Exceptions in Java**

**1. Checked Exceptions**

* Checked at **compile time**.
* Must be either caught or declared to be thrown.
* Examples:
  + IOException
  + SQLException
  + FileNotFoundException

**2. Unchecked Exceptions (Runtime Exceptions)**

* Checked at **runtime**.
* Generally caused by programming bugs.
* Examples:
  + ArithmeticException
  + NullPointerException
  + ArrayIndexOutOfBoundsException

**3. Errors**

* Serious problems beyond the control of the application (e.g., memory issues).
* Example:
  + OutOfMemoryError
  + StackOverflowError

**Java Exception Hierarchy**

Throwable

├── Error (not usually caught)

└── Exception

   ├── Checked Exceptions

   └── RuntimeException (Unchecked Exceptions)

**Java Exception Handling Keywords**

**1. try**

Contains code that might throw an exception.

try {

   // code that might throw an exception

}

**2. catch**

Handles the exception thrown in try.

catch (ExceptionType name) {

   // handling code

}

**3. finally (Optional block)**

Contains code that **always executes**, whether or not an exception occurs.

finally {

   // cleanup code

}

**4. throw**

Used to **explicitly throw** an exception.

throw new ArithmeticException("Division by zero");

**5. throws**

Used in method declarations to specify which exceptions can be thrown.

public void readFile() throws IOException {

   // method code

}

**Example**

public class ExceptionExample {

   public static void main(String[] args) {

       try {

           int data = 10 / 0;  // throws ArithmeticException

       } catch (ArithmeticException e) {

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

       } finally {

          System.out.println("Finally block always executes.");

       }

   }

}

**Output:**

Exception caught: java.lang.ArithmeticException: / by zero

Finally block always executes.

**Multiple Catch Blocks**

You can catch multiple exceptions using multiple catch blocks.

try {

   int a = 50 / 0;

} catch (ArithmeticException e) {

  System.out.println("Arithmetic Exception");

} catch (Exception e) {

  System.out.println("General Exception");

}

Note: Catch the most specific exceptions first (child classes).

**Custom Exceptions**

You can create your own exception class by extending Exception or RuntimeException.

class MyException extends Exception {

   MyException(String message) {

      super(message);

   }

}

**Usage:**

public class TestCustomException {

   static void checkAge(int age) throws MyException {

       if (age < 18)

           throw new MyException("Not eligible to vote");

   }

   public static void main(String[] args) {

       try {

          checkAge(16);

       } catch (MyException e) {

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

       }

   }

}

Example==========

// Custom checked exception  
class InvalidAgeException extends Exception {

    // Default constructor  
    public InvalidAgeException() {  
        super("Invalid age provided");  
    }

    // Constructor with custom message  
    public InvalidAgeException(String message) {  
        super(message);  
    }  
}  
class AgeValidator {  
    public void validate(int age) throws InvalidAgeException {  
        if (age < 18) {  
            throw new InvalidAgeException("Age must be at least 18 to cast a vote!!!!");  
        }  
    }

    public static void main(String[] args) {  
        AgeValidator validator = new AgeValidator();  
        try {  
            validator.validate(16);  
        } catch (InvalidAgeException e) {  
            System.out.println("Caught exception: " + e.getMessage());  
        }  
    }  
}

**Best Practices**

* Catch specific exceptions instead of generic ones.
* Use finally for releasing resources like file streams or DB connections.
* Don't suppress exceptions silently.
* Avoid using exceptions for control flow.
* Always close resources (try-with-resources is helpful here).

**try-with-resources (Java 7+)**

Automatically closes resources like files.

try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {

   String line = br.readLine();

} catch (IOException e) {

  e.printStackTrace();

}

**Summary**

| **Keyword** | **Purpose** |
| --- | --- |
| try | Block to monitor for exceptions |
| catch | Handles specific exceptions |
| finally | Executes regardless of exception occurrence |
| throw | Explicitly throws an exception |
| throws | Declares exceptions a method may throw |

class Throw\_Throws {

    public static void checkAge(int age) throws IllegalArgumentException{  
        if (age < 18) {  
            throw new IllegalArgumentException("Access denied – You must be at least 18 years old.");  
        } else {  
            System.out.println("Access granted – You are eligible to cast a vote!");  
        }  
    }

    public static void main(String[] args) {  
        try {  
            checkAge(19);    
        } catch (IllegalArgumentException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }

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

=============================================================

**What Is Exception Propagation in Java?**

In **Java**, when an exception occurs and is not caught in the current method, it is **automatically passed (propagated)** to the method that called it. This continues **up the call stack** until the exception is either:

1. **Caught** using a try-catch block, or
2. **Terminates the program** if not handled anywhere.

**Java Example:**

public class ExceptionPropagation {

   public static void methodA() {

       methodB();

   }

   public static void methodB() {

       methodC();

   }

   public static void methodC() {

       // Exception occurs here

       int result = 10 / 0;

   }

   public static void main(String[] args) {

       methodA();

   }

}

**What Happens Here:**

* methodC() causes a **divide by zero** (ArithmeticException).
* There is **no try-catch**, so it **propagates** to methodB(), then to methodA(), then to main().
* Since no one handles it, the program crashes and prints a **stack trace**.

**Key Points:**

* **Propagation** = passing the exception to the calling method.
* Works for both **checked** and **unchecked** exceptions.
* You can use throws in method signatures to allow exceptions to propagate for **checked exceptions**.

**Example with Checked Exception (IOException):**

import java.io.\*;

public class Example {

   public static void methodA() throws IOException {

       methodB();

   }

   public static void methodB() throws IOException {

       throw new IOException("File not found");

   }

   public static void main(String[] args) {

       try {

           methodA();

       } catch (IOException e) {

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

       }

   }

}

***NOTE: In this case, the exception is declared with throws, and main() handles it.***

**Multithreading**

**What is Multithreading in Java?**

Multithreading in Java is a feature that allows the concurrent execution of two or more parts of a program for maximum utilization of CPU.

IMPORTANT POINT:

**“Each part of such a program is called a thread,** **and each thread defines a separate path of execution.”**

Multithreading is used to perform multiple tasks simultaneously, such as:

* Performing background tasks without interrupting the main program.
* Speeding up complex computations by dividing them across threads.

**Concepts**

* **Thread**: A lightweight subprocess; the smallest unit of processing.
* **Main thread**: When a Java program starts, one thread (main thread) is created.

**Java Thread Lifecycle**

**1. New (Created)**

* A thread is created but not yet started.
* Example:

Thread t = new Thread(); // A new Thread by the name “t” is created.

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

**2. Runnable state**

* “t.start();”
* After you call start(), the thread moves to the **Runnable** state.
* It is ready to run but may or may not be running immediately (depends on thread scheduler).

**Example:**

t.start(); // internally t.start() method calls the run() method

* The thread scheduler picks a thread from the **Runnable pool** to run.

**3. Running state**

* The thread is actively executing its run() method.
* This is technically part of the Runnable state, but here the thread is currently executing.

**4. Blocked / Waiting / Timed Waiting**

* A thread can move to these states when:
  + It’s waiting for a resource (e.g., waiting to acquire a lock).
  + It’s waiting indefinitely (wait() method).
  + It’s sleeping (sleep() method).
  + It’s waiting for a specific time (join(timeout)).

**Examples:**

Thread.sleep(1000);        // Timed waiting  1 second

synchronized(lock) { ... } // Could cause blocking

lock.wait();               // Waiting

t.join();                 // Waiting for another thread to finish

**5. Terminated (Dead)**

* The thread finishes its task or is stopped.
* Once the run() method completes or the thread is stopped (deprecated), it reaches this state.
* The thread cannot be restarted once terminated.
* Example:

// After run() completes, thread is terminated

**Creating Threads**: In Java, threads can be created by:

1. **Extending the Thread class --🡪 class MyThread extends Thread**
2. **Implementing the Runnable interface --🡪 class MyThread implements Runnable**

**Example 1: Using Thread class**

class MyThread extends Thread {

**public void run() {**

       for (int i = 1; i <= 5; i++) {

           System.out.println(Thread.currentThread().getName() + " - Value: " + i);

           try {

              Thread.sleep(1000); // Pause for 1 second

           } catch (InterruptedException e) {

              System.out.println(e);

           }

       }

   }

    public static void main(String[] args) {

       MyThread t1 = new MyThread();

       MyThread t2 = new MyThread();

       t1.setName("Thread-1");

       t2.setName("Thread-2");

       t1.start(); // Start first thread

       t2.start(); // Start second thread

   }

}

**Example 2: Using Runnable interface**

class MyRunnable implements Runnable {

   public void run() {

       for (int i = 1; i <= 5; i++) {

           System.out.println(Thread.currentThread().getName() + " - Value: " + i);

           try {

              Thread.sleep(2000);

           } catch (InterruptedException e) {

              System.out.println(e);

           }

       }

   }

   public static void main(String[] args) {

       MyRunnable runnable = new MyRunnable();

       Thread t1 = new Thread(runnable, "Thread-1");

       Thread t2 = new Thread(runnable, "Thread-2");

//The Runnable interface does not have a start() method, so we require to create the object of the Thread class

       t1.start();

       t2.start();

   }

}

**Benefits of Multithreading**

* **Improved performance** on multi-core systems.
* **Better resource utilization**.
* Enables **asynchronous** programming (non-blocking).

class Counter {  
    private int count = 0;  
    // synchronized method to avoid race condition  
    public synchronized void increment() {  
        //System.out.print("Count: "+count);  
    count++;  
  }

    public int getCount() {  
        return count;  
    }  
}

class SyncThread extends Thread {  
    Counter counter;

    SyncThread(Counter counter) {  
        this.counter = counter;  
    }

    public void run() {  
        System.out.println("Run method");  
    for (int i = 0; i < 1000; i++) {  
            counter.increment();  
        }  
    }

    public static void main(String[] args) throws InterruptedException {  
        Counter counter = new Counter();

        SyncThread t1 = new SyncThread(counter);  
        SyncThread t2 = new SyncThread(counter);

        t1.start();  
        t2.start();  
  
    t1.join();  
    t2.join();  
  
  
        System.out.println("Final Count: " + counter.getCount());   
    }  
}

Controlling Threads

**Controlling threads in Java using:**

* sleep() – to pause a thread
* join() – to make one thread wait for another
* isAlive() – to check if a thread is still running
* synchronized – to avoid concurrency issues

**Example: Thread Control with sleep(), join(), isAlive()**

class ControlledThread extends Thread {

   public void run() {

       for (int i = 1; i <= 3; i++) {

          System.out.println(getName() + " - Count: " + i);

           try {

              Thread.sleep(1000); // pause for 1 second

           } catch (InterruptedException e) {

              System.out.println(e);

           }

       }

   }

   public static void main(String[] args) {

       System.out.println(“Main thread starts….”);

ControlledThread t1 = new ControlledThread();

      ControlledThread t2 = new ControlledThread();

      t1.setName("Worker-1");

      t2.setName("Worker-2");

      System.out.println("Starting threads...");

       t1.start();

       t2.start();

        try {

           // Wait for t1 to finish before continuing

           t1.join();

       } catch (InterruptedException e) {

          System.out.println(e);

       }

       // Check if threads are alive

      System.out.println("Is " + t1.getName() + " alive? " + t1.isAlive());

      System.out.println("Is " + t2.getName() + " alive? " + t2.isAlive());

      System.out.println("Main thread finished.");

   }

}

**Output (sample)**

Starting threads...

Worker-1 - Count: 1

Worker-2 - Count: 1

Worker-1 - Count: 2

Worker-2 - Count: 2

Worker-1 - Count: 3

Worker-2 - Count: 3

Is Worker-1 alive? false

Is Worker-2 alive? false

Main thread finished.

**Example: Using synchronized to control access to shared resource**

class Counter {

   private int count = 0;

   // synchronized method to avoid race condition

   public synchronized void increment() {

       count++;

   }

   public int getCount() {

       return count;

   }

}

class SyncThread extends Thread {

   Counter counter;

   SyncThread(Counter counter) {

       this.counter = counter;

   }

   public void run() {

       for (int i = 0; i < 1000; i++) {

          counter.increment();

       }

   }

   public static void main(String[] args) throws InterruptedException {

       Counter counter = new Counter();

       SyncThread t1 = new SyncThread(counter);

       SyncThread t2 = new SyncThread(counter);

       t1.start();

       t2.start();

       t1.join();

       t2.join();

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

   }

}

**Use Both:**

1. synchronized ensures **correct access** to shared data but **does not control thread timing or order**.
2. join() is about **thread lifecycle coordination** — waiting for threads to complete before proceeding.

**Inter-thread communication**

**Inter-thread communication** in Java allows **multiple threads** to communicate and cooperate with each other by **waiting** and **notifying**. It’s especially useful when threads share resources and must coordinate access or execution.

Java provides 3 key methods (from Object class):

| **Method** | **Purpose** |
| --- | --- |
| wait() | Tells the current thread to release the lock and go to sleep until another thread calls notify() or notifyAll() |
| notify() | Wakes up one thread waiting on the same object's monitor |
| notifyAll() | Wakes up all threads waiting on the same object |

**Example: Producer-Consumer Problem (Inter-thread Communication)**

This is a classic example to demonstrate inter-thread communication using wait() and notify().

**Code:**

class SharedData {

   private int data;

   private boolean hasData = false;

   // Producer puts data

   public synchronized void produce(int value) throws InterruptedException {

       while (hasData) {

           wait(); // wait if data is already produced

       }

       data = value;

       hasData = true;

      System.out.println("Produced: " + data);

       notify(); // notify consumer

   }

// Consumer gets data

   public synchronized void consume() throws InterruptedException {

       while (!hasData) {

           wait(); // wait if no data is available

       }

      System.out.println("Consumed: " + data);

       hasData = false;

       notify(); // notify producer

   }

}

class Producer extends Thread {

   SharedData shared;

  Producer(SharedData shared) {

       this.shared = shared;

   }

   public void run() {

       int value = 1;

       try {

           for (int i = 0; i < 5; i++) {

              shared.produce(value++);

              Thread.sleep(500); // simulate time delay

           }

       } catch (InterruptedException e) {

          System.out.println(e);

       }

   }

}

class Consumer extends Thread {

   SharedData shared;

  Consumer(SharedData shared) {

       this.shared = shared;

   }

   public void run() {

       try {

           for (int i = 0; i < 5; i++) {

              shared.consume();

              Thread.sleep(1000); // simulate processing delay

           }

       } catch (InterruptedException e) {

          System.out.println(e);

       }

   }

}

public class InterThreadCommunicationExample {

   public static void main(String[] args) {

       SharedData shared = new SharedData();

      Producer p = new Producer(shared);

       Consumer c = new Consumer(shared);

       p.start();

       c.start();

   }

}

**Output**

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

Produced: 3

Consumed: 3

...

**Key Points:**

* wait() causes the **current thread to wait** until another thread **notifies** it.
* notify() or notifyAll() should be called on the **same object** the thread is waiting on.
* Important point :These methods must be called **inside a synchronized block/method**.

**Thread Priority in Java:**

* Java assigns thread priorities using **integers from 1 to 10**, defined by constants:
  + Thread.MIN\_PRIORITY = 1
  + Thread.NORM\_PRIORITY = 5 (default)
  + Thread.MAX\_PRIORITY = 10

**How to Set Thread Priority:**

Thread t = new Thread();

t.setPriority(Thread.MAX\_PRIORITY); // Sets the highest priority

You can also set it directly:

t.setPriority(8);

**Get Thread Priority:**

int p = t.getPriority();

System.out.println("Priority: " + p);

**Important:**

* **Thread priority is just a suggestion to the JVM** and OS.
* **Thread scheduling is platform-dependent**, so priority **may not guarantee** execution order.

----------

class SharedData {  
    private int data;  
    private boolean hasData = false;  
    // Producer puts data  
    public synchronized void produce(int value) throws InterruptedException {  
        while (hasData) {  
            wait(); // wait if data is already produced  
        }  
        data = value;  
        hasData = true;  
        System.out.println("Produced: " + data);  
        notify(); // notify consumer  
    }

// Consumer gets data  
    public synchronized void consume() throws InterruptedException {  
        while (!hasData) {  
            wait(); // wait if no data is available  
        }  
        System.out.println("Consumed: " + data);  
        hasData = false;  
        notify(); // notify producer  
    }  
}  
class Producer extends Thread {  
    SharedData shared;  
    Producer(SharedData shared) {  
        this.shared = shared;  
    }  
    public void run() {  
        int value = 1;  
        try {  
            for (int i = 0; i < 5; i++) {  
                shared.produce(value++);  
                Thread.sleep(500); // simulate time delay  
            }  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
}

class Consumer extends Thread {  
    SharedData shared;

    Consumer(SharedData shared) {  
        this.shared = shared;  
    }

    public void run() {  
        try {  
            for (int i = 0; i < 5; i++) {  
                shared.consume();  
                Thread.sleep(1000); // simulate processing delay  
            }  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }  
    }  
}

class Impl{  
    public static void main(String[] args) {  
        SharedData shared = new SharedData();  
       Producer p = new Producer(shared);  
        Consumer c = new Consumer(shared);  
        p.start();  
        c.start();  
    }  
}

1. Which method must be implemented when you implement Runnable?  
   A) start( )  
   **B) run( )**  
   C) execute( )  
   D) thread( )
2. Which method starts a new thread in Java?  
   **A) start( )**  
   B) run( )  
   C) execute( )  
   D) init( )
3. \*\*\*\*\*\*What does Thread.sleep(1000) do?  
   **A) Releases the lock for 1 second**  
   B) Suspends the thread execution for 1 second  
   C) Terminates the thread  
   D) Waits for another thread
4. Which method is used to pause current thread until another thread finishes?  
   A) wait( )  
   B) notify( )  
   **C) join( )**  
   D) sleep( )
5. What does isAlive() method check?  
   A) Whether thread has started and not yet terminated  
   B) Whether thread is sleeping  
   **C) Whether thread is dead**  
   D) None of the above
6. Which keyword is used for synchronization in Java?  
   A) locked  
   B) mutex  
   **C) synchronized**  
   D) volatile
7. What is the result of modifying a non-volatile shared variable in one thread but reading it in another without synchronization?  
   A) Always latest value  
   **B) Race condition**  
   C) Compile-time error  
   D) Deadlock
8. \*\*\*\*\*\*\*Which interface extends Runnable and supports returning a result?  
   **A) Callable**  
   B) Future  
   C) Supplier  
   D) Executor
9. What is the output order of start() vs run()?  
   A) run() invokes in new thread  
   B) start() executes in current thread  
   **C) run() executes directly in current thread**  
   D) start() directly calls run()
10. Using Thread t = new Thread(runnable); t.run();…  
    A) Creates a new thread  
    **B) Runs on the same (main) thread**  
    C) Causes compile error  
    D) Runs asynchronously
11. Which method wakes only a single waiting thread?  
    A) notifyAll( )  
    B) wake( )  
    **C) notify( )**  
    D) signal( )
12. Which method wakes all waiting threads?  
    A) notify( )  
    B) busy( )  
    **C) notifyAll( )**  
    D) broadcast( )
13. What's the typical use case of volatile keyword?  
    A) Atomic operations  
    **B) Guarantee visibility across threads**  
    C) Locking mechanism  
    D) Scheduling threads
14. What happens if notify() is called but no thread is waiting?  
    A) Exception  
    **B) No effect**  
    C) Deadlock  
    D) Block current thread
15. If two threads call synchronized method on same object, what happens?  
    A) Both execute concurrently  
    **B) One executes, one blocks**  
    C) Both block  
    D) Exception
16. For different synchronized methods in same object, is lock shared?  
    **A) Yes**  
    B) No  
    C) Only on static methods  
    D) Depends on thread priority
17. Which yields higher throughput?  
    **A) notify() over notifyAll()**  
    B) notifyAll() over notify()  
    C) Both equal  
    D) Neither is good
18. \*\*\*\*\*\*What does Thread.yield() request?  
    A) Sleep  
    **B) Give up CPU momentarily**  
    C) Terminate thread  
    D) None
19. What is deadlock?  
    **A) Threads blocked forever waiting for each other**  
    B) Thread blocked waiting data  
    C) Multiple threads sleeping  
    D) Thread throwing exception
20. Which mechanism helps avoid deadlock?  
    A) Thread.yield( )  
    **B) Ordering lock acquisitions consistently**  
    C) notifyAll( )  
    D) wait( )
21. wait() must be used within…  
    A) runnable  
    **B) synchronized block on same object**  
    C) static method  
    D) main method
22. What exception thrown if wait() is called outside synchronized?  
    **A) IllegalMonitorStateException**  
    B) IllegalStateException  
    C) InterruptedException  
    D) NullPointerException
23. Which method interrupts a thread?  
    A) kill()  
    B) stop()  
    **C) interrupt()**  
    D) terminate()
24. Thread.currentThread().getPriority( ) default is…  
    A) 1  
    **B) 5**  
    C) 10  
    D) OS default
25. Which is true about Thread priorities?  
    A) Always respected  
    **B) May be ignored by JVM, every OS has a different JVM(JVM is platform dependent)**  
    C) Causes thread pooling  
    D) Determines memory allocation
26. interrupt() on sleeping thread…  
    A) Continues sleeping  
    **B) InterruptedException thrown**  
    C) Silent fail  
    D) Terminates process

---------------

**Multithreading with StringBuffer**

* StringBuffer is a mutable, thread-safe class for working with strings.
* It is synchronized, meaning multiple threads can access it safely, but one at a time.

**Example:**

In the following example, multiple threads will **append characters** to a shared StringBuffer object.

**Code:**

class StringBufferTask extends Thread {

   private StringBuffer buffer;

   private String text;

   public StringBufferTask(StringBuffer buffer, String text) {

       this.buffer = buffer;

       this.text = text;

   }

   public void run() {

       synchronized (buffer) {

           for (int i = 0; i < text.length(); i++) {

              buffer.append(text.charAt(i));

               try {

                  Thread.sleep(50); // Simulate delay

               } catch (InterruptedException e) {

                  System.out.println(e);

               }

           }

          System.out.println(Thread.currentThread().getName() + " appended: " + text);

       }

   }

    public static void main(String[] args) {

       StringBuffer sharedBuffer = new StringBuffer();

       StringBufferTask t1 = new StringBufferTask(sharedBuffer, "Hello");

      StringBufferTask t2 = new StringBufferTask(sharedBuffer, "World");

      StringBufferTask t3 = new StringBufferTask(sharedBuffer, "Java");

       t1.start();

       t2.start();

       t3.start();

       try {

           t1.join();

           t2.join();

           t3.join();

       } catch (InterruptedException e) {

          System.out.println(e);

       }

      System.out.println("Final content: " + sharedBuffer.toString());

   }

}

**Output:**

Thread-0 appended: Hello

Thread-2 appended: Java

Thread-1 appended: World

Final content: HelloJavaWorld

***Note: Output order may vary due to thread scheduling.***

OPERATORS:

**Precedence of Operators**

| **Precedence Level** | **Operators** | **Associativity** | **Description** |
| --- | --- | --- | --- |
| **1 (highest)** | [], (), . | Left to Right | Array access, method call, member access |
| **2** | ++, -- (postfix) | Left to Right | Post-increment/decrement |
| **3** | ++, --, +, -, ~, ! | Right to Left | Unary plus/minus, bitwise NOT, logical NOT |
| **4** | \*, /, % | Left to Right | Multiplication, division, modulo |
| **5** | +, - | Left to Right | Addition, subtraction (also string concat) |
| **6** | <<, >>, >>> | Left to Right | Bitwise shift |
| **7** | <, <=, >, >=, instanceof | Left to Right | Comparison, type comparison |
| **8** | ==, != | Left to Right | Equality |
| **9** | & | Left to Right | Bitwise AND |
| **10** | ^ | Left to Right | Bitwise XOR |
| **11** | ` | ` | Left to Right |
| **12** | && | Left to Right | Logical AND |
| **13** | ` |  | ` |
| **14** | ?: | Right to Left | Ternary conditional |
| **15** | =, +=, -=, \*=, /=, %=, &=, ` | =, ^=, <<=, >>=, >>>=` | Right to Left |
| **16 (lowest)** | , | Left to Right | Comma operator |

**Wrapper Classes**

**Wrapper classes are used to convert primitive data types into objects.** Each primitive type has a corresponding wrapper class in the “java.lang package”.

**Why Wrapper Classes?**

Java is an object-oriented language, but primitive types (like int, double) are not objects. There are situations where objects are required, such as:

* When working with collections like ArrayList, which only store objects (not primitives).
* For utility methods in classes like Integer, Double, etc.
* For using features like autoboxing and unboxing.

**Primitive Types vs Wrapper Classes**

| **Primitive Type** | **Wrapper Class** |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| Long | Long |
| Float | Float |
| double | Double |
| Char | Character |
| boolean | Boolean |

**Example**

int num = 10;                      // primitive data type

Integer obj = Integer.valueOf(num); // manual **boxing**

int x = obj.intValue();           // manual **unboxing**

**Autoboxing and Unboxing (Automatic Conversion)**

**Java 5+ supports autoboxing and unboxing**, which automatically converts between primitives and their wrappers.

Integer obj = 5;    // autoboxing: int -> Integer

int num = obj;      // auto-unboxing: Integer -> int

**Use Case: Collections**

ArrayList<Integer> list = new ArrayList<>();

list.add(10);  // autoboxed from int to Integer

***NOTE: We can't use ArrayList<int> because collections require objects.***

**Common Methods in Wrapper Classes**

Integer.parseInt("123");  // Converts String to int

Double.parseDouble("3.14"); //autoboxing

Boolean.parseBoolean("true");

**These are static utility methods provided by wrapper classes.**

**Summary**

* Wrapper classes "wrap" primitive types in objects.
* They allow primitives to be used in object-oriented features (like collections).
* Java supports autoboxing/unboxing to simplify their use.
* **Each primitive type has a corresponding wrapper class.**

**Constructors and Methods of Wrapper classes**

**Constructors of Wrapper Classes**

Most wrapper classes (except Character and Boolean) have constructors that are **deprecated** as of Java 9 because autoboxing and static methods like valueOf() are preferred.

**Example: Deprecated Constructors**

Integer intObj = new Integer(10);          // Deprecated

Double doubleObj = new Double("3.14");     // Deprecated

**Use valueOf() instead:**

Integer intObj = Integer.valueOf(10);      // Preferred

**2. Common Static Methods**

***All wrapper classes provide useful static methods for conversion and parsing.***

**valueOf()**

***Converts a primitive or a String to a wrapper object.***

Integer i = Integer.valueOf(100);

Double d = Double.valueOf("12.34");

**parse<Type>()**

***Converts a String to a primitive.***

int i = Integer.parseInt("123");       // returns int

double d = Double.parseDouble("3.14");

boolean b = Boolean.parseBoolean("true");

**3. Instance Methods**

Wrapper objects provide methods to get the **primitive value**, **compare values**, or **convert to Strings**.

***<datatype>Value() methods***

***Returns the primitive value of the object.***

Integer i = 42;

int primitive = i.intValue();     // returns int 42

double d = i.doubleValue();      // returns 42.0

***compareTo()***

**Compares two wrapper objects.**

Integer a = 10;

Integer b = 20;

int result = a.compareTo(b); // returns -1 (because 10 < 20)

***equals(Object obj)***

***Checks for value equality.***

Integer a = 100;

Integer b = 100;

System.out.println(a.equals(b));  // true

**toString()**

Converts the object to a String.

Integer a = 50;

String s = a.toString(); // "50"

**4. Special Methods in Some Classes**

**Character class:**

* isDigit(char ch)
* isLetter(char ch)
* toLowerCase(char ch)
* toUpperCase(char ch)

char c = 'A';

System.out.println(Character.isLetter(c));  // true

**Boolean class:**

* parseBoolean(String s)
* valueOf(String s)
* booleanValue()

class WClass{  
  public static void main(String[] args){  
    int x=10;  
    // auto-boxing - Converting primitive data type to wrapper class object.   
    //Integer ibox = x;  
    //Integer ibox = 40;    
    Integer ibox = Integer.valueOf(x);   
    System.out.println("Autoboxing: "+ibox);  
  
    // auto-unboxing - Converting back the wrapper class object  
    //           to the primitive data type.    
    int unbox = ibox;  
    System.out.println("Auto-Unboxing: "+unbox);  
  
    //Constructors  
    Integer intObj = new Integer(10);// Deprecated  
    System.out.println("Deprecated using Constructor: "+ intObj);  
  
    //Use valueOf() method  
    Integer vintObj = Integer.valueOf(10);  
    System.out.println("Use valueOf() method instead: "+vintObj);  
  
    String str ="100";  
    int istr = Integer.parseInt(str);  
    System.out.println("Convert a string to an integer: "+(istr+100));  
  
    //Returns the primitive value of the object.  
    Integer objInt = 300;  
    int primInt = objInt.intValue();  
    System.out.println("The int value for a wrapperclass object: "+primInt);  
  
    //floating type wrapper class object  
    Float objFloat = 23.45f;  
    float primFloat = objFloat.floatValue();  
    System.out.println("The float value for a wrapperclass object: "+primFloat);  
  
    //compareTo()   
    //returns 0 when 2 wrapper class objects are equal.  
    // returns 1 if first WC object is greater than the second.  
    // returns -1 if first WC object is less than the second    
    Integer a = 100;  
    Integer b = 100;  
    int result = a.compareTo(b);  // returns -1 (because 10 < 20)  
    System.out.println("Comapring 2 wrapper class objects: "+result);  
  
    //equals(Object obj)  
    //Checks for value equality.  
    Integer obj1 = 100;  
    Integer obj2 = 100;  
    System.out.println("Checks value equality for WC objects: "+(obj1.equals(obj2)));  
  
    //toString() - Converts the object to a String.  
    Integer a1 = 50;  
    String s = a1.toString();  // "50"  
    System.out.println("Converting the WC object to a String value: "+(s+100));  
  
    //isLetter()  
    char c = 'A';  
    System.out.println(Character.isLetter(c));  // true  
  
    //isDigit()  
    char ch = '2';  
    System.out.println(Character.isDigit(ch));

    char theChar='V';  
    char lCase = Character.toLowerCase(theChar);  
    System.out.println("To LowerCase: "+lCase);  
  
    char uCase = Character.toUpperCase(theChar);  
    System.out.println("To UpperCase: "+uCase);  
  
  
  }

}

**MCQ – Wrapper classes**

1. **Which of the following is not a wrapper class in Java?**  
   a) Integer  
   b) Double  
   **c) String**  
   d) Boolean
2. **What package contains the wrapper classes in Java?**  
   **a) java.lang**  
   b) java.util  
   c) java.io  
   d) java.wrapper
3. **What is the wrapper class for int?**  
   **a) Integer**  
   b) Int  
   c) intObject  
   d) Number
4. **Which of the following is the correct way to create an Integer object?**  
   a) Integer i = 10;  
   b) Integer i = new Integer(10);  
   c) Integer i = Integer.valueOf(10);  
   **d) All of the above**
5. **Autoboxing is:**  
   a) Converting wrapper to primitive  
   **b) Converting primitive to wrapper**  
   c) A type of inheritance  
   d) None of the above
6. **Unboxing refers to:**  
   **a) Converting wrapper to primitive**  
   b) Converting primitive to object  
   c) Boxing a class  
   d) None of the above
7. **Which of the following is immutable in Java?**  
   a) String  
   b) Integer  
   c) Boolean  
   **d) All of the above**

1. **What is returned by Integer.parseInt("123")?**  
   a) "123"  
   b) Integer object  
   **c) int primitive 123**  
   d) Exception
2. **What happens if you parse a non-numeric string using Integer.parseInt("abc")?**  
   a) Returns 0  
   b) Returns null  
   **c) Throws NumberFormatException**  
   d) Compiles but does nothing
3. **Which method returns the double value of a Double object?**  
   a) getDouble()  
   **b) doubleValue()**  
   c) valueOf()  
   d) parseDouble()
4. **valueOf() in wrapper classes returns:**  
   a) A primitive  
   **b) A wrapper object**  
   c) An array  
   d) A string
5. **Which method is static in Integer class?**  
   a) intValue()  
   **b) parseInt(String s)**  
   c) doubleValue()  
   d) None
6. **Which method converts Boolean to primitive boolean?**  
   a) getBoolean()  
   **b) booleanValue()**  
   c) valueOf()  
   d) toBoolean()
7. **Which of these is not a valid wrapper class constructor?**  
   a) new Integer("10")  
   b) new Double(10.5)  
   c) new Boolean(true)  
   **d) new Character("c")**
8. **Which wrapper class method compares two objects numerically?**  
   a) equals()  
   **b) compareTo()**  
   c) toString()  
   d) valueOf()
9. **Which version of Java introduced autoboxing?**  
   a) JDK 1.2  
   b) JDK 1.3  
   **c) JDK 1.5**  
   d) JDK 1.8
10. **Autoboxing allows conversion from:**  
    a) Wrapper to primitive  
    **b) Primitive to wrapper**  
    c) Object to primitive  
    d) String to wrapper
11. **Select the option:**

Integer i = 10;

int a = i;

a) Invalid  
b) Autoboxing  
**c) Unboxing**  
d) Compilation error

1. **Which of the following is unboxing?**  
   **a) int a = new Integer(5);**  
   b) Integer i = 5;  
   c) Integer i = new Integer(5);  
   d) int a = 5;
2. **Which wrapper class supports character data type?**  
   a) Char  
   b) String  
   **c) Character**  
   d) Text
3. **Which is more memory-efficient: primitive or wrapper class?**  
   **a) Primitive**  
   b) Wrapper  
   c) Both same  
   d) Depends on context

1. **Wrapper classes are mainly used for:**  
   a) UI design  
   b) Multithreading  
   **c) Working with collections**  
   d) None of the above
2. **What is the result of Integer.valueOf("abc")?**  
   a) 0  
   b) "abc"  
   **c) Throws NumberFormatException**  
   d) null
3. **new Integer(10).equals(10) returns:**  
   **a) true**  
   b) false  
   c) Compile error  
   d) Exception
4. **Which is not a wrapper class method?**  
   a) valueOf()  
   b) toString()  
   c) intValue()  
   **d) get()**
5. **Which of these does not support autoboxing?**  
   a) int  
   b) double  
   **c) void**  
   d) char
6. **What is the output of Integer.compare(3, 5)?**  
   **a) -1**  
   b) 1  
   c) 0  
   d) Error
7. **What is returned by Boolean.parseBoolean("true")?**  
   **a) true**  
   b) false  
   c) Boolean object  
   d) null
8. **Which one is not overloaded in Integer class?**  
   a) parseInt()  
   b) valueOf()  
   **c) equals()**  
   d) toBinaryString()
9. **Wrapper classes extend which class?**  
   a) Object  
   b) Number (except Character and Boolean)  
   c) Wrapper  
   **d) a and b**

**Generics**

**What are Generics?**

Generics enable **classes, interfaces, and methods** to operate on **typed parameters**—this means more reusable and type-safe code.

**Example: Generic Class for a Container**

Let's create a generic class Box that can store any type of object.

// Generic class

class Box<T> {

   private T item;

   public void set(T item) {

       this.item = item;

   }

   public T get() {

       return item;

   }

}

public class Main {

   public static void main(String[] args) {

       // Box for Integer

      Box<Integer> intBox = new Box<>();

      intBox.set(100);

      System.out.println("Integer value: " + intBox.get());

       // Box for String

      Box<String> strBox = new Box<>();

      strBox.set("Implementing Generics");

      System.out.println("String value: " + strBox.get());

 // Box for Double

Box<Double> dVar = new Box<>();

dvar.set(3543.33);

System.out.println("Double value: “+dvar.get());

}

}

**Output**

Integer value: 100

String value: Generics in Java

**Benefits**

* No need to write separate classes for Integer, String, etc.
* Type safety: You can't accidentally store a String in a Box<Integer>.

**2. Use of Wrapper Classes in Java**

**What are Wrapper Classes?**

Wrapper classes convert **primitive data types** into **objects** so they can be used in:

* Generics (which require objects)
* Collections like ArrayList
* Utility methods (like parsing Strings)

| **Primitive** | **Wrapper Class** |
| --- | --- |
| Int | Integer |
| Double | Double |
| Char | Character |
| Boolean | Boolean |

**Example: Using Wrapper Classes with Generics (ArrayList)**

import java.util.ArrayList;

public class WrapperExample {

   public static void main(String[] args) {

      ArrayList<Integer> numbers = new ArrayList<>();

       // Autoboxing: int → Integer

      numbers.add(10);

      numbers.add(20);

      numbers.add(30);

       int sum = 0;

       for (Integer num : numbers) {

           // Unboxing: Integer → int

           sum += num;

       }

      System.out.println("Sum: " + sum);

   }

}

**Output**

Sum: 60

**Understand the concept**

* int values are **autoboxed** into Integer objects.
* The generic ArrayList<Integer> stores these objects.
* On retrieval, they’re **unboxed** back into int.

**Wrapping It All Up**

| **Feature** | **Description** | **Example** |
| --- | --- | --- |
| **Generics** | Create reusable, type-safe classes/methods | Box<T>, ArrayList<T> |
| **Wrapper Classes** | Convert primitives to objects for generic use | Integer, Double, etc. |
| **Autoboxing** | Java converts primitive to wrapper automatically | list.add(10); // int → Integer |
| **Unboxing** | Java converts wrapper to primitive automatically | int x = list.get(0); |

**Arrays**

**Arrays in Java are the objects that store the elements in contiguous locations.**

***Arrays are fixed in size.***

**Example:**

int[] arr = new int[4];

arr[5]=42362; //Incorrect

**Collection and Collections Framework**

**What is Collection in Java?**

In Java, **Collection** is a **root interface** in the **java.util package**. It represents a **group of objects**, known as **elements**. The Collection interface is the foundation of the **Java Collections Framework**, which is used for storing, retrieving, and manipulating data in data structures like lists, sets, and queues.

**Key Points about Collection:**

* It is part of the **Java Collections Framework**.
* It defines the **basic operations** like adding, removing, and checking elements.
* It is a **generic interface**: Collection<E>, where E is the type of elements.
* Commonly used through its **subinterfaces**:
  + List – ordered collection (e.g., ArrayList)
  + Set – no duplicates (e.g., HashSet)
  + Queue – ordered for processing (e.g., PriorityQueue)

**What is the Collections Framework in Java?**

The **Java Collections Framework** is a **unified architecture** for storing, managing, and manipulating groups of **objects**. It is part of the java.util package and provides **interfaces**, **classes**, and **algorithms** to work with different types of data structures like lists, sets, queues, and maps.

**Key Components of the Collections Framework:**

**1. Interfaces (Blueprints)**

Define the standard ways to work with collections:

| **Interface** | **Description** |
| --- | --- |
| Collection<E> | Root interface of the framework |
| List<E> | Ordered collection (duplicates allowed) |
| Set<E> | Unordered collection (no duplicates) |
| Queue<E> | Collection designed for holding elements prior to processing |
| Deque<E> | Double-ended queue |
| Map<K, V> | Key-value pairs (not a true subinterface of Collection) |

**2. Implementations (Concrete Classes)**

These are actual data structure classes that implement the interfaces:

| **Interface** | **Implementation Classes** |
| --- | --- |
| List | ArrayList, LinkedList, Vector |
| Set | HashSet, LinkedHashSet, TreeSet |
| Queue | PriorityQueue, ArrayDeque, LinkedList |
| Map | HashMap, TreeMap, LinkedHashMap, Hashtable |
| Collections Framework = Data Structures + Algorithms |  |

**3. Algorithms (via Collections utility class)**

The **java.util.Collections** class provides **static methods** for tasks like:

* Sorting: Collections.sort(list)
* Shuffling: Collections.shuffle(list)
* Finding min/max: Collections.max(set)
* Reversing: Collections.reverse(list)

**4. Iterator and Iterable**

***Used to loop over collections:***

Iterator<String> it = list.iterator();

while(it.hasNext()) {

   System.out.println(it.next());

}

**Benefits of Using Collections Framework**

* **Standardized API**: All collections follow the same rules and interfaces.
* **Code Reusability**: Write general code for any type of collection.
* **Efficiency**: Well-optimized data structures.
* **Type Safety** (Generics): Prevents runtime type errors.

**Example:**

import java.util.\*;

public class FrameworkExample {

   public static void main(String[] args) {

      List<String> names = new ArrayList<>();

       names.add("Alice");

       names.add("Bob");

       names.add("Charlie");

      Collections.sort(names);

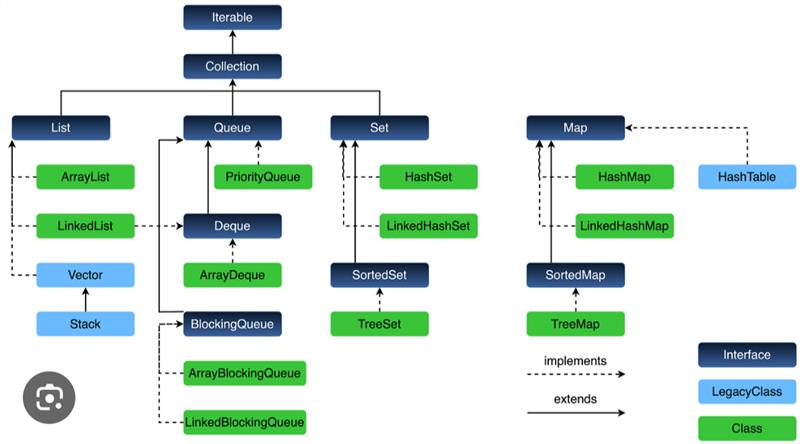
      System.out.println("Sorted: " + names);

   }

}

**Summary**

| **Concept** | **Description** |
| --- | --- |
| **Collections Framework** | Complete set of interfaces, implementations, and utility classes for managing data structures. |
| **Includes** | List, Set, Map, Queue, Collections class |
| **Purpose** | To provide reusable, efficient, and flexible tools for handling data. |



//Step 1  
import java.util.\*;  
class FrameworkExample {  
    public static void main(String[] args) {  
        //Step 2 - create an object of the ArrayList bclass  
    List<String> names = new ArrayList<>();  
  
    //Step 3 - Add elements to the list  
    names.add("Steven");  
        names.add("Bob");  
        names.add("Alex");  
    names.add("Alex");  
    names.add("Turner");  
    names.add("Martin");  
    names.add("Fox");  
    names.add("Jack");  
    names.add("Lily");  
    //names.add(null);  
  
    System.out.println("Before the sorting......"+names);

        Collections.sort(names);  
        System.out.println("After the Sorting: " + names);  
    }  
}

**List Interface**

**What is List in Java Collections Framework?**

In Java, **List is an interface** **in the Collections Framework**, found in the java.util package. It extends the Collection interface and represents an ordered collection (also known as a sequence).

interface List extends Collection{}

**Key Features of List:**

* **Ordered:** Maintains insertion order.
* **Indexed:** Elements can be accessed by their index.
* **Allows duplicates.**
* **Supports null values** (depends on implementation).
* Common implementations:
  + **ArrayList** (resizable array)
  + **LinkedList** (doubly-linked list)
  + **Vector** (synchronized version of ArrayList)
  + **Stack** (LIFO stack)

**Common Methods of List Interface**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| add(E e) | Adds an element to the list | list.add("Apple"); |
| add(int index, E element) | Inserts element at specified position | list.add(1, "Banana"); |
| get(int index) | Returns element at index | list.get(0); |
| set(int index, E element) | Replaces element at index | list.set(0, "Grapes"); |
| remove(int index) | Removes element at index | list.remove(1); |
| remove(Object o) | Removes the first occurrence of the object | list.remove("Apple"); |
| indexOf(Object o) | Returns first index of element | list.indexOf("Apple"); |
| lastIndexOf(Object o) | Returns last index of element | list.lastIndexOf("Apple"); |
| contains(Object o) | Checks if element exists | list.contains("Apple"); |
| isEmpty() | Checks if list is empty | list.isEmpty(); |
| size() | Returns number of elements | list.size(); |
| clear() | Removes all elements | list.clear(); |
| subList(int fromIndex, int toIndex) | Returns a view (slice) of part of the list | list.subList(1, 3); |
| iterator() | Returns an iterator for the list | list.iterator(); |

**Example Code Using ArrayList:**

import java.util.\*;

public class ListExample {

   public static void main(String[] args) {

      List<String> fruits = new ArrayList<>();

      fruits.add("Apple");

      fruits.add("Banana");

      fruits.add("Cherry");

      fruits.add("Apple");

      System.out.println("Original List: " + fruits);

      System.out.println("Element at index 1: " + fruits.get(1));

       fruits.set(2, "Blueberry");

      System.out.println("After set: " + fruits);

      fruits.remove("Apple"); // Removes first "Apple"

      System.out.println("After removing 'Apple': " + fruits);

      System.out.println("Index of Banana: " + fruits.indexOf("Banana"));

      System.out.println("Size: " + fruits.size());

      System.out.println("Sublist (0 to 2): " + fruits.subList(0, 2));

      fruits.clear();

      System.out.println("Is empty? " + fruits.isEmpty());

   }

}

                                                           -------------------------

import java.util.\*;  
class ListExample {  
    public static void main(String[] args) {  
        List<String> fruits = new ArrayList<>();  
        fruits.add("Apple");  
        fruits.add("Banana");  
        fruits.add("Cherry");  
        fruits.add("Apple");  
  
    System.out.println("Original List: " + fruits);  
        System.out.println("Element at index 1: " + fruits.get(1));  
        fruits.set(2, "Blueberry");  
        System.out.println("After set: " + fruits);  
        fruits.remove("Apple"); // Removes first "Apple"  
        System.out.println("After removing 'Apple': " + fruits);  
        System.out.println("Index of Banana: " + fruits.indexOf("Banana"));  
        System.out.println("Size: " + fruits.size());  
        System.out.println("Sublist (0 to 2): " + fruits.subList(0, 2));  
        fruits.clear();  
    System.out.println("Empty List: "+fruits);  
        System.out.println("Is empty? " + fruits.isEmpty());  
    }  
}

import java.util.ArrayList;  
import java.util.Iterator;

class ArrayListIteratorExample {  
    public static void main(String[] args) {  
        // Create an ArrayList  
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements to the ArrayList  
        fruits.add("Apple");  
        fruits.add("Banana");  
        fruits.add("Mango");  
        fruits.add("Orange");

      /\*   
          Creating a reference of type Iterator<String>,   
      which points to an object that is an instance of a    
           class that implements the Iterator interface.  
           \*/  
    Iterator<String> iterator = fruits.iterator();  
        // Traverse the ArrayList using Iterator  
        System.out.println("Fruits in the list:");  
        while (iterator.hasNext()) {  
            String fruit = iterator.next();  
            System.out.println(fruit);  
    }      
  
    //Enhanced for loop  
    for(String favfruits:fruits){  
      System.out.println(favfruits);  
    }  
  
  }   
}

import java.util.ArrayList;  
import java.util.Iterator;

class ArrayListIteratorExample {  
    public static void main(String[] args) {  
        // Create an ArrayList  
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements to the ArrayList  
        fruits.add("Apple");  
        fruits.add("Banana");  
        fruits.add("Mango");  
        fruits.add("Orange");

      /\*   
          Creating a reference of type Iterator<String>,   
      which points to an object that is an instance of a    
           class that implements the Iterator interface.  
           \*/  
    Iterator<String> iterator = fruits.iterator();  
        // Traverse the ArrayList using Iterator  
        System.out.println("Fruits in the list:");  
        while (iterator.hasNext()) {  
            String fruit = iterator.next();  
            System.out.println(fruit);  
    }      
  
    //Enhanced for loop  
    for(String favfruits:fruits){  
      System.out.println(favfruits);  
    }  
  
  }   
}

**LinkedList**

In **Java's Collections Framework**, a **LinkedList** is a class that implements both the **List** and **Deque** interfaces. It is part of the java.util package.

**LinkedList features:**

* A **doubly linked list** implementation of the List and Deque interfaces.
* Allows **duplicates**.
* Maintains **insertion order**.
* Efficient for **frequent insertions/deletions**, especially at the beginning or middle.

**Declaration:**

import java.util.LinkedList;

LinkedList<String> list = new LinkedList<>();

**Common LinkedList Methods (with Examples):**

**1. add() – Adds element**

list.add("Apple");            // Add at end

list.add(1, "Banana");        // Add at index

list.addFirst("Start");       // Add at beginning

list.addLast("End");          // Add at end

**2. get() – Access element**

String item = list.get(1);   // Get element at index 1

list.getFirst();             // First element

list.getLast();              // Last element

**3. set() – Replace element**

list.set(1, "Orange");        // Replace index 1 with "Orange"

**4. remove() – Remove elements**

list.remove();               // Remove first

list.remove(2);              // Remove by index

list.remove("Banana");        // Remove by value

list.removeFirst();          // Remove first element

list.removeLast();           // Remove last element

**5. size() – List size**

int size = list.size();

**6. contains() – Check if element exists**

boolean exists = list.contains("Apple");

**7. clear() – Remove all elements**

list.clear();

**8. isEmpty() – Check if list is empty**

boolean empty = list.isEmpty();

**9. iterator() – Traverse list**

for (String item : list) {

  System.out.println(item);

}

OR

Iterator<String> it = list.iterator();

while(it.hasNext()) {

   String item = it.next();

  System.out.println(item);

}

**Summary**

| **Feature** | **LinkedList** |
| --- | --- |
| Type | Doubly Linked List |
| Allows duplicates? | Yes |
| Maintains order? | Yes (insertion order) |
| Implements | List, Deque, Queue |
| Performance | Fast insert/delete, slow random access |

                                                                          --------------------

 import java.util.LinkedList;  
import java.util.ListIterator;

class ListIteratorExample {  
    public static void main(String[] args) {  
        LinkedList<String> list = new LinkedList<>();  
        list.add("A");  
        list.add("B");  
        list.add("C");

        ListIterator<String> iterator = list.listIterator();

        // Traverse forward  
        while (iterator.hasNext()) {  
            System.out.println("Next: " + iterator.next());  
        }

        // Traverse backward  
        while (iterator.hasPrevious()) {  
            System.out.println("Previous: " + iterator.previous());  
        }

        // Modify during iteration  
        while (iterator.hasNext()) {  
            String element = iterator.next();  
            if (element.equals("B")) {  
                iterator.set("Bee");        // Replace "B" with "Bee"  
                iterator.add("D");          // Add "D" after "Bee"  
            }  
        }

        System.out.println("Modified list: " + list);  
    }  
}

**Set Interface**

In Java's **Collections Framework**, the **Set** interface is a part of the java.util package and represents a collection that **does not allow duplicate elements**. It models the mathematical set abstraction.

**Key Features of Set Interface:**

* **No duplicates** allowed.
* **At most one null** element is allowed (in some implementations like HashSet).
* **Unordered** by default (unless using a sorted or ordered implementation).

**Common Implementations of Set:**

| **Implementation** | **Ordering** | **Thread Safety** | **Performance** | **Allows null?** |
| --- | --- | --- | --- | --- |
| HashSet | No order | No | Fast | Yes (1 null) |
| LinkedHashSet | Insertion order | No | Slightly slower than HashSet | Yes |
| TreeSet | Sorted (natural or custom comparator) | No | Slower than HashSet | No (throws NullPointerException) |

**Set Interface Hierarchy:**

java.util.Collection

       ↳ java.util.Set

**Example:**

import java.util.\*;

public class SetExample {

   public static void main(String[] args) {

      Set<String> fruits = new HashSet<>();

       fruits.add("Apple");

       fruits.add("Banana");

       fruits.add("Apple"); // Duplicate - won't be added

       fruits.add("Orange");

       System.out.println("Fruits set: " + fruits);

   }

}

**Common Methods in Set:**

| **Method** | **Description** |
| --- | --- |
| add(E e) | Adds the specified element |
| remove(Object o) | Removes the specified element |
| contains(Object o) | Returns true if the element exists |
| size() | Returns the number of elements |
| isEmpty() | Checks if the set is empty |
| clear() | Removes all elements |
| iterator() | Returns an iterator for traversal |

import java.util.\*;  
class SetExample {  
    public static void main(String[] args) {  
        Set<String> fruits = new HashSet<>();  
        fruits.add("Apple");  
        fruits.add("Banana");  
        fruits.add("Orange");  
    fruits.add("Blueberry");  
        fruits.add("Grapes");  
        fruits.add("Guava");  
        fruits.add(null);  
    fruits.add(null);  
    fruits.add(null);  
    fruits.add("Mango");  
        fruits.add("PineApple");  
        fruits.add("Papaya");  
    fruits.add("Pomegranate");  
  
    //No compilation error will be shown  
    fruits.add("Apple"); // Duplicate - won't be added  
        fruits.add("Apple"); // Duplicate - won't be added  
        fruits.add("Apple"); // Duplicate - won't be added  
        fruits.add("Apple"); // Duplicate - won't be added  
        fruits.add("Apple"); // Duplicate - won't be added  
  
  
        System.out.println("Fruits set: " + fruits);  
    }  
}

**HashSet**

The HashSet class in Java is part of the **Collections Framework** and implements the **Set** interface backed by a **HashMap**. It is used to **store unique elements** and allows **at most one null value**.

**Key Features of HashSet:**

* **No duplicates**
* **Unordered** (no guarantee of insertion order)
* **Allows one null** value
* **Not synchronized** (not thread-safe)
* **Faster** than TreeSet and LinkedHashSet for basic operations like add(), remove(), contains()

**HashSet Class Declaration:**

public class HashSet<E> extends AbstractSet<E>

   implements Set<E>, Cloneable, java.io.Serializable

**Example: HashSet**

import java.util.HashSet;

public class HashSetExample {

   public static void main(String[] args) {

       // Creating a HashSet of Strings

      HashSet<String> cities = new HashSet<>();

       // Adding elements

       cities.add("Delhi");

       cities.add("Mumbai");

       cities.add("Kolkata");

       cities.add("Delhi"); // Duplicate - will not be added

       cities.add(null);    // null is allowed

       cities.add(null);    // Duplicate null - ignored

       // Displaying the HashSet

      System.out.println("Cities in HashSet: " + cities);

       // Checking existence

      System.out.println("Contains Mumbai? " + cities.contains("Mumbai"));

       // Removing an element

       cities.remove("Kolkata");

      System.out.println("After removing Kolkata: " + cities);

   }

}

**Common Operations and Their Time Complexity:**

| **Operation** | **Time Complexity** |
| --- | --- |
| add() | O(1) average |
| remove() | O(1) average |
| contains() | O(1) average |
| iteration | O(n) |

**Iterator in CollectionsFramework**

**What is an Iterator in Java?**

* **java.util.Iterator** is a **generic interface** that provides methods for iterating over collections.
* It is part of the **Java Collections Framework**.
* It provides a way to **traverse (loop through)** a collection like ArrayList, HashSet, etc., **one element at a time**.

It's part of the java.util package and is useful when you want to:

* Go through each element in a collection.
* Optionally remove elements while looping.
* Avoid using a traditional for-loop (especially when collection type may change).

**Decalration:**

*public interface Iterator<E> {*

*boolean hasNext();*

*E next();*

*void remove(); // optional*

*}*

**Example: Using ArrayList with Iterator**

import java.util.ArrayList;

import java.util.Iterator;

class ArrayListIteratorExample {

   public static void main(String[] args) {

       // Create an ArrayList

      ArrayList<String> fruits = new ArrayList<>();

       // Add elements to the ArrayList

       fruits.add("Apple");

      fruits.add("Banana");

      fruits.add("Mango");

      fruits.add("Orange");

/\*

              Creating a reference of type Iterator<String>, which points to an object that is an instance of a

              class that implements the Iterator interface.

          \*/

**Iterator<String> iterator = fruits.iterator();**

       // Traverse the ArrayList using Iterator

      System.out.println("Fruits in the list:");

       while (iterator.hasNext()) {

           String fruit = iterator.next();

          System.out.println(fruit);

       }    } }

**HashSet**

HashSet again a part of Java's Collections Framework and implements the Set interface. It is used to create a collection that does not allow duplicate elements. (Because it is a part of Set interface)

Internally, HashSet uses a **HashMap** to store its elements.

**Key Features of HashSet**

| **Feature** | **Description** |
| --- | --- |
| No duplicates | Only unique elements are allowed |
| No ordering | Elements are not stored in the order they were inserted |
| Allows null | Only one null element is allowed |
| Not synchronized | Use Collections.synchronizedSet() for thread safety |
| **NOTE: Backed by HashMap** | **Uses hashing mechanism for storage** |

**Class Declaration**

public class HashSet<E> extends AbstractSet<E>

**implements Set<E>,** Cloneable, Serializable

**Commonly Used Constructors**

HashSet()                         // Default initial capacity (16) and load factor (0.75)

HashSet(int initialCapacity)     // Set initial capacity

HashSet(int initialCapacity, float loadFactor)

HashSet(Collection<? extends E> c) // Create set with elements from a collection

**Common Methods in HashSet**

| **Method** | **Description** |
| --- | --- |
| add(E e) | Adds the specified element |
| remove(Object o) | Removes the specified element |
| contains(Object o) | Returns true if element exists |
| size() | Returns number of elements |
| isEmpty() | Checks if the set is empty |
| clear() | Removes all elements |
| iterator() | Returns an iterator over elements |
| clone() | Returns a shallow copy |
| **toArray()** | **Converts the HashSet to an array** |

**Example of HashSet**

import java.util.\*;

public class HashSetExample {

   public static void main(String[] args) {

       // Create HashSet

      HashSet<String> set = new HashSet<>();

       // Add elements

      set.add("Apple");

      set.add("Banana");

       set.add("Orange");

      set.add("Apple"); // duplicate, will not be added

       // Print HashSet

      System.out.println("HashSet: " + set);

       // Check if element exists

      System.out.println("Contains 'Banana'? " + set.contains("Banana"));

       // Remove an element

      set.remove("Banana");

       // Size of set

      System.out.println("Size: " + set.size());

      // Iterate using for-each

       for (String fruit : set) {

          System.out.println(fruit);

       }

       // Clear the set

       set.clear();

              System.out.println(fruits);

      System.out.println("Is empty? " + set.isEmpty());

   }

}

**How HashSet Works**

* **HashSet uses a HashMap to store elements.**
* When you call add(element), it actually does:

map.put(Banana, PRESENT);

* Here, PRESENT is a dummy constant value.

**Example: HashSet with Custom Objects**

To use custom objects in a HashSet, **override equals() and hashCode() methods.**

import java.util.\*;

class Student {

   private int id;

   private String name;

   public Student(int id, String name) {

       this.id = id;

       this.name = name;

   }

// Override hashCode and equals

**@Override**

**public int hashCode() {**

**return Objects.hash(id, name);**

**}**

**@Override**

**public boolean equals(Object obj) {**

**if (this == obj) return true;**

**if (!(obj instanceof Student)) return false;**

**Student other = (Student) obj;**

**return id == other.id && Objects.equals(name, other.name);**

**}**

   @Override

   public String toString() {

       return id + " - " + name;

   }

}

public class CustomHashSet {

   public static void main(String[] args) {

      HashSet<Student> students = new HashSet<>();

      students.add(new Student(1, "Steven"));

      students.add(new Student(2, "Harry"));

      students.add(new Student(1, "Harry")); // Duplicate

       for (Student s : students) {

          System.out.println(s);

       }

   }

}

**Limitations of HashSet**

* **No ordering**: Use LinkedHashSet if order matters.
* **Not thread-safe**: Use Collections.synchronizedSet() for thread safety.
* **Poor performance if hashCode() is not well-distributed**.

B**est Practices**

* Always override equals() and hashCode() in custom classes.
* Use TreeSet if you need sorted elements

**How HashSet Works**

* **HashSet uses a HashMap to store elements.**
* When you call add(element), it actually does:

map.put(Banana, PRESENT);

* Here, PRESENT is a dummy constant value.

**Example: HashSet with Custom Objects**

To use custom objects in a HashSet, **override equals() and hashCode() methods.**

import java.util.\*;

class Student {

   private int id;

   private String name;

   public Student(int id, String name) {

       this.id = id;

       this.name = name;

   }

// Override hashCode and equals

**@Override**

**public int hashCode() {**

**return Objects.hash(id, name);**

**}**

**@Override**

**public boolean equals(Object obj) {**

**if (this == obj) return true;**

**if (!(obj instanceof Student)) return false;**

**Student other = (Student) obj;**

**return id == other.id && Objects.equals(name, other.name);**

**}**

   @Override

   public String toString() {

       return id + " - " + name;

   }

}

public class CustomHashSet {

   public static void main(String[] args) {

      HashSet<Student> students = new HashSet<>();

      students.add(new Student(1, "Steven"));

      students.add(new Student(2, "Harry"));

      students.add(new Student(1, "Harry")); // Duplicate

       for (Student s : students) {

          System.out.println(s);

       }

   }

}

**Limitations of HashSet**

* **No ordering**: Use LinkedHashSet if order matters.
* **Not thread-safe**: Use Collections.synchronizedSet() for thread safety.
* **Poor performance if hashCode() is not well-distributed**.

B**est Practices**

* Always override equals() and hashCode() in custom classes.
* Use TreeSet if you need sorted elements.

package pack1;

import java.util.HashSet;  
import java.util.Objects;

class Person {  
    private String name;  
    private int age;

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

    // Override hashCode() and equals()  
    @Override  
    public int hashCode() {  
        return Objects.hash(name, age); // Combines both fields  
    }

    @Override  
    public boolean equals(Object obj) {  
        if (this == obj) return true;  
        if (!(obj instanceof Person)) return false;  
        Person p = (Person) obj;  
        return age == p.age && Objects.equals(name, p.name);  
    }

@Override  
    public String toString() {  
        return name + " (" + age + ")";  
    }  
}

public class HashCodeDemo {  
    public static void main(String[] args) {  
        HashSet<Person> people = new HashSet<>();

        Person p1 = new Person("Harry", 25);  
        Person p2 = new Person("Steven", 30);  
        Person p3 = new Person("Harry", 25); // Same as p1

        people.add(p1);   // p1 is sharing a different memory loc  
        people.add(p2);  
        people.add(p3); // Will NOT be added (duplicate) // // p3 is sharing a different memory loc

        System.out.println("People in HashSet:");  
        for (Person p : people) {  
            System.out.println(p + " - hashCode: " + p.hashCode());  
        }  
    }  
}

**Why hashCode() is Important in HashSet**

In Java, HashSet uses the **hashCode() and equals()** methods to determine whether two objects are the **same** (i.e., duplicates).

* hashCode() is used to find the *bucket* in which the object may be stored.
* equals() is used to compare the actual content for equality.

**Example: hashCode() in a Custom Class with HashSet**

import java.util.HashSet;

import java.util.Objects;

class Person {

   private String name;

   private int age;

   Person(String name, int age) {

       this.name = name;

       this.age = age;

   }

   // Override hashCode() and equals()

   @Override

   public int hashCode() {

       return Objects.hash(name, age); // Combines both fields

   }

   @Override

   public boolean equals(Object obj) {

       if (this == obj) return true;

       if (!(obj instanceof Person)) return false;

       Person p = (Person) obj;

       return age == p.age && Objects.equals(name, p.name);

   }

@Override

   public String toString() {

       return name + " (" + age + ")";

   }

}

public class HashCodeDemo {

   public static void main(String[] args) {

      HashSet<Person> people = new HashSet<>();

       Person p1 = new Person("Harry", 25);

       Person p2 = new Person("Steven", 30);

       Person p3 = new Person("Harry", 25); // Same as p1

      people.add(p1);   // p1 is sharing a different memory loc

      people.add(p2);

      people.add(p3); // Will NOT be added (duplicate) // // p3 is sharing a different memory loc

      System.out.println("People in HashSet:");

       for (Person p : people) {

          System.out.println(p + " - hashCode: " + p.hashCode());

       }

   }

}

**Output**

People in HashSet:

P1-Steven(25) - hashCode: 356573597

P2=Harry(30) - hashCode: 1735600054

P3 =Harry(25) is only added once, even though p3 is a separate object with the same data.

**Explanation**

* Even though p1 and p3 are two **different instances**, hashCode() and equals() make them behave as the **same** in a HashSet.
* If you **don't override** hashCode() and equals(), both objects will be treated as different — leading to **duplicates**.

**What Happens Without hashCode() and equals()?**

If you remove or comment out those methods:

@Override

public int hashCode() { ... }

@Override

public boolean equals(Object obj) { ... }

**IMPORTANT:**

***Then both p1 and p3 would be stored in the HashSet as different elements, because the default Object methods compare by memory address.***

**TreeSet**

A TreeSet in Java is a **NavigableSet** implementation based on a **Red-Black Tree** (a self-balancing binary search tree). It stores elements in **sorted (ascending)** order and **does not allow duplicates**.

**What is a Red-Black Tree?**

**A Red-Black Tree is a binary search tree (BST)** with an added set of coloring rules (red or black) that ensure the tree remains balanced for insert, delete, and search operations.

**TreeSet Class Declaration**

public class TreeSet<E> extends AbstractSet<E>

       implements NavigableSet<E>, Cloneable, Serializable

**Key Features of TreeSet**

| **Feature** | **Description** |
| --- | --- |
| Sorted | Maintains elements in natural order |
| No Duplicates | Like other Set implementations |
| Null Handling | Does **not** allow null in non-empty sets (throws NullPointerException) |
| Not Thread-Safe | Use Collections.synchronizedSortedSet() if needed |
| Backed by | **Red-Black Tree** |

**Constructors**

TreeSet()                             // Natural ordering

TreeSet(Comparator<? super E> comp)  // Custom comparator

TreeSet(Collection<? extends E> c)   // Elements from a collection

TreeSet(SortedSet<E> s)              // Elements from another sorted set

**Basic Example**

import java.util.TreeSet;

public class TreeSet {

   public static void main(String[] args) {

      TreeSet<Integer> numbers = new TreeSet<>();

      numbers.add(40);

      numbers.add(10);

      numbers.add(30);

      numbers.add(20);

      System.out.println("TreeSet: " + numbers);

      numbers.remove(30);

      System.out.println("After removal: " + numbers);

   }

}

**Output:**

TreeSet: [10, 20, 30, 40]

After removal: [10, 20, 40]

**Common TreeSet Methods**

| **Method** | **Description** |
| --- | --- |
| add(E e) | Adds element in sorted position |
| remove(Object o) | Removes specified element |
| contains(Object o) | Checks if element exists |
| size() | Returns the number of elements |
| clear() | Empties the set |
| first() / last() | Returns first or last (smallest/largest) element |
| ceiling(E e) | Least element ≥ given element |
| ceil(floor(E e) | Greatest element ≤ given element |
| higher(E e) | Strictly greater element |
| lower(E e) | Strictly smaller element |
| headSet(E to) | Elements < given value |
| tailSet(E from) | Elements ≥ given value |
| subSet(E from, E to) | Elements between values |
| descendingSet() | Returns elements in descending order |
| iterator() | Returns iterator in ascending order |
| descendingIterator() | Iterator in descending order |

**Example with Navigable Methods**

import java.util.TreeSet;

public class TreeSetNavigation {

   public static void main(String[] args) {

      TreeSet<Integer> tree = new TreeSet<>();

       tree.add(10);

       tree.add(20);

       tree.add(30);

       tree.add(40);

      System.out.println("TreeSet: " + tree);

      System.out.println("First: " + tree.first());       // 10

      System.out.println("Last: " + tree.last());         // 40

      System.out.println("Higher than 20: " + tree.higher(20)); // 30

      System.out.println("Lower than 20: " + tree.lower(20));   // 10

      System.out.println("Ceiling of 25: " + tree.ceiling(25)); // 30

      System.out.println("Floor of 25: " + tree.floor(25));     // 20

   }

}

**Comparator Interface:**

**The Comparator interface in Java is part of the java.util package and is used to define custom sorting logic for objects.**

**It allows you to compare two objects** **of the same type** to determine their order.

**Purpose:**

* To **control the order of objects** in a collection (like List, Set, Map) **without modifying the actual class** of those objects.
* It is useful when:
  + You can't modify the object's class to implement Comparable
  + You need multiple different ways to sort the same type of objects

**Syntax:**

//Functional Interface

public interface Comparator<T> {

   int compare(T o1, T o2);

}

* Returns:
  + **Negative int** if o1 < o2
  + **Zero** if o1 == o2
  + **Positive int** if o1 > o2

**Example:**

import java.util.\*;

class Student {

   private String name;

   private int age;

   Student(String name, int age) {

       this.name = name;

       this.age = age;

   }

   public String toString() {

       return name + " (" + age + ")";

   }

}

**//FIRST APPROACH**

class AgeComparator **implements Comparator<Student>** {

   public int compare(Student s1, Student s2) {

       return Integer.compare(s1.age, s2.age);

   }

}

public class Main {

   public static void main(String[] args) {

      List<Student> list = new ArrayList<>();

       list.add(new Student("Harry", 18));

       list.add(new Student("Garry", 23));

       list.add(new Student("Rocky", 16));

      Collections.sort(list, new AgeComparator());

      System.out.println(list);

   }

}

**When to Use Comparator?**

Use Comparator when:

* You want **multiple ways to sort** (e.g., by name, salary, or ID).
* You **can't modify the original class** (e.g., the class is from a library).
* You want to separate comparison logic from the object itself.

**Define an Employee class, and two comparators:**

* SortBySalary (ascending)
* SortByName (alphabetical)

**Employee.java**

class Employee {

   int id;

   String name;

   double salary;

   Employee(int id, String name, double salary) {

       this.id = id;

       this.name = name;

       this.salary = salary;

   }

   public String toString() {

       return id + " - " + name + " - $" + salary;

   }

}

import java.util.Comparator;

class SortBySalary implements Comparator<Employee> {

   public int compare(Employee e1, Employee e2) {

       return Double.compare(e1.salary, e2.salary);

   }

}

class SortByName implements Comparator<Employee> {

   public int compare(Employee e1, Employee e2) {

       return e1.name.compareTo(e2.name);

   }

}

**Example**

import java.util.ArrayList;

import java.util.Collections;

public class ComparatorExample {

   public static void main(String[] args) {

      ArrayList<Employee> list = new ArrayList<>();

       list.add(new Employee(101, "Jack", 5000));

       list.add(new Employee(102, "Garry", 3000));

       list.add(new Employee(103, "Brown", 6000));

      System.out.println("Original List:");

       for (Employee e : list) {

          System.out.println(e);

       }

**// Sort by salary**

       Collections.sort(list, new SortBySalary());

      System.out.println("\nSorted by Salary:");

       for (Employee e : list) {

          System.out.println(e);

       }

**// Sort by name**

Collections.sort(list, new SortByName());

      System.out.println("\nSorted by Name:");

       for (Employee e : list) {

          System.out.println(e);

       }

   }

}

Approaches:

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;

class Student {  
  private String name;  
  private int age;  
  
  public Student() {  
    super();  
    // TODO Auto-generated constructor stub  
  }

  public Student(String name, int age) {  
    super();  
    this.name = name;  
    this.age = age;  
  }  
  
  public String getName() {  
    return name;  
  }

  public int getAge() {  
    return age;  
  }

  @Override  
  public String toString() {  
    return "Student [name=" + name + ", age=" + age + "]";  
  }  
  
}

class StudentMain {

  public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    List<Student> list = new ArrayList<>();  
    list.add(new Student("Garry",21));  
    list.add(new Student("Binny",26));  
    list.add(new Student("Fenny",22));  
    /\*  
    //FirstApproach  
    //SortByAge  
    System.out.println("Before sorting: The age of the Students");  
    for(Student sage:list) {  
      System.out.println(sage.getAge());  
    }  
  
    System.out.println("After sorting: The age of the Students");  
    Collections.sort(list,new AgeComparator());  
    for(Student sage:list) {  
      System.out.println(sage.getAge());  
    }  
  
    //SortByName  
    System.out.println("Before sorting: The name of the Students");  
    for(Student name:list) {  
      System.out.println(name.getName());  
    }  
  
    System.out.println("After sorting: The name of the Students");  
    Collections.sort(list, new NameComparator());  
    for(Student name:list) {  
      System.out.println(name.getName());  
    }\*/  
  /\*  
    //Second Approach  
    //Using Anonymous class  
    System.out.println("After sorting: The age of the Students...");  
    Collections.sort(list,new Comparator<Student>() {  
      public int compare(Student s1,Student s2) {  
        return Integer.compare(s1.getAge(), s2.getAge());  
      }});  
  
    for(Student sage:list) {  
      System.out.println(sage.getAge());  
    }  
    \*/  
  
    //Third Approach  
    //Using Lambda Expression  
    list.sort((s1,s2)->Integer.compare(s1.getAge(),s2.getAge()));  
  
    // To iterate the elements of the ArrayList  
    list.forEach(System.out::println);  
  
  }

}

**Comparable Interface**

The Comparable interface is used to define the **natural ordering** of objects. A class that implements Comparable must override the compareTo() method.

**Syntax:**

public class MyClass implements Comparable<MyClass> {

   @Override

   public int compareTo(MyClass other) {

       // return negative, zero, or positive

   }

}

**Rules:**

* Returns:
  + negative → this < other
  + zero → this == other
  + positive → this > other

**Example: Using Comparable**

import java.util.\*;

class Student implements Comparable<Student> {

   private int id;

   private String name;

   Student(int id, String name) {

       this.id = id;

       this.name = name;

   }

    // Natural ordering by id

   public int compareTo(Student s) {

              return this.id - s.id;

   }

   public String toString() {

       return this.id + " " + this.name;

   }

}

public class Main {

   public static void main(String[] args) {

      List<Student> list = new ArrayList<>();

       list.add(new Student(3, "Garry"));

       list.add(new Student(1, "Jane"));

       list.add(new Student(2, "Jackson"));

Collections.sort(list);

// Bydefault the sort() static method of Collections class internally calls    //compareTo() method //to compare the 2 objects.

       for (Student s : list) {

          System.out.println(s);

       }

   }

}

| **Feature** | **Comparable** | **Comparator** |
| --- | --- | --- |
| Interface in package | java.lang | java.util |
| Method | int compareTo(T o) | int compare(T o1, T o2) |
| Used for | Natural ordering | Custom ordering |
| Modify source class? | Yes (implements Comparable) | No (can sort externally) |
| Can define multiple? | No (only one compareTo) | Yes (multiple comparators for flexibility) |
| Java 8+ enhancements | None | Lambdas, Comparator.comparing(), etc. |

//Natural sorting usinmg Comparable

import java.util.List;  
import java.util.ArrayList;  
import java.util.Collections;  
class Student implements Comparable<Student> {  
    private int id;  
    private String name;  
    Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }

    // Natural ordering by id  
    public int compareTo(Student s) {  
  return this.id - s.id;     
    }

    public String toString() {  
        return this.id + " " + this.name;  
    }  
}

class Main {  
    public static void main(String[] args) {  
        List<Student> list = new ArrayList<>();  
        list.add(new Student(3, "Garry"));  
        list.add(new Student(1, "Jane"));  
        list.add(new Student(2, "Jackson"));

  Collections.sort(list);   
// Bydefault the sort() static method of Collections class internally calls    //compareTo() method //to compare the 2 objects.

        for (Student s : list) {  
            System.out.println(s);  
        }  
    }  
}

**MCQ – CollectionsFramework**

1. Which interface allows duplicate elements and maintains insertion order?  
   a) Set b) List c) Queue d) Map  
   **Answer:** b
2. ArrayList is backed by a:  
   a) Linked list b) Array c) Hash table d) Balanced tree  
   **Answer:** b
3. Which List implementation is best for frequent random access?  
   a) LinkedList b) ArrayList c) Vector d) Stack  
   **Answer:** b
4. Which List implementation is optimized for frequent insertions/removals at both ends?  
   a) ArrayList b) Vector c) LinkedList d) CopyOnWriteArrayList  
   **Answer:** c
5. Calling list.get(i) on a LinkedList runs in:  
   a) O(1) b) O(log n) c) O(n) d) O(n²)  
   **Answer:** c
6. To ensure thread safety, you should use:  
   a) ArrayList b) LinkedList c) Vector d) HashSet  
   **Answer:** c
7. Which method on ArrayList doubles its capacity when needed?  
   a) ensureCapacity() b) grow() c) resize() d) expand()  
   **Answer:** b (internally grow())
8. Which supports null elements?  
   a) ArrayList b) LinkedList c) HashSet d) All of the above  
   **Answer:** d
9. add(index, element) in ArrayList runs in:  
   a) O(1) b) O(log n) c) O(n) d) O(n log n)  
   **Answer:** c
10. The default capacity of a new ArrayList is:  
    a) 8 b) 10 c) 16 d) 0  
    **Answer:** b
11. Which interface prevents duplicate elements?  
    a) List b) Set c) Collection d) Iterable  
    **Answer:** b
12. HashSet does **not** guarantee:  
    a) No duplicates b) Constant-time add/remove c) Insertion order d) Null element support  
    **Answer:** c
13. TreeSet keeps elements in:  
    a) Insertion order b) Sorted order c) Reverse insertion order d) Random order  
    **Answer:** b
14. If two objects are “equal” via equals(), they must have equal:  
    a) hashCode() b) size c) compareTo() d) toString()  
    **Answer:** a
15. What structure underlies TreeSet?  
    a) Hash table b) Linked list c) Binary search tree d) Balanced tree (Red‑Black)  
    **Answer:** d
16. To iterate in insertion order with no duplicates, use:  
    a) TreeSet b) HashSet c) LinkedHashSet d) ArrayList  
    **Answer:** c
17. If you add duplicates to a Set, what happens?  
    a) Throws exception b) Keeps all c) Keeps first or last only d) Ignores duplicates  
    **Answer:** d
18. TreeSet requires that elements be:  
    a) Serializable b) Comparable or Comparator provided c) Cloneable d) Iterable  
    **Answer:** b
19. Which yields higher lookup performance for HashSet?  
    a) O(log n) b) O(n) c) O(1) average d) O(n log n)  
    **Answer:** c
20. A null element in TreeSet will cause:  
    a) It works fine b) Runtime exception c) Ignored safely d) Stored at root  
    **Answer:** b (NullPointerException for comparison)
21. Comparable<T> defines a method:  
    a) compare(a, b) b) compareTo(T o) c) equals(T o) d) hashCode()  
    **Answer:** b
22. compareTo() should return > 0 if the current object is:  
    a) Less than b) Equal c) Greater than d) None  
    **Answer:** c
23. If a.compareTo(b) == 0, then a and b are:  
    a) Equal by equals() necessarily b) Possibly equal c) Not equal d) Always different  
    **Answer:** b (should be consistent with equals but not enforced)
24. Which signature correctly implements Comparable?  
    a) class A implements Comparable<A> { public int compareTo(A o) {...} }  
    b) class A implements Comparable { public int compare(Object o) {...} }  
    c) class A implements Comparable<A> { public void compareTo(A o) {...} }  
    d) class A implements Comparable<?> { … }  
    **Answer:** a
25. When sorting with Collections.sort(list), the list elements must implement:  
    a) Serializable b) Cloneable c) Comparable(compareTo) d) Comparator  
    **Answer:** c
26. Violating compareTo() contract may cause:  
    a) Logical errors in sorting/searching b) Compile errors c) Runtime casting errors d) Memory leaks  
    **Answer:** a
27. Which return value means “this < other”?  
    a) Positive b) Zero c) Negative d) Infinity  
    **Answer:** c
28. Comparable is located in package:  
    a) java.util b) java.lang c) java.io d) java.lang.reflect  
    **Answer:** b
29. \*\*\*\*\*\*\*\*Built‑in Comparable support:  
    a) String b) Integer c) Date d) All of the above  
    **Answer:** d
30. Natural ordering means ordering defined by:  
    a) Comparator b) equals() c) compareTo() d) hashCode()  
    **Answer:** c (Collections.sort(list);
31. Comparator<T> defines:  
    a) compareTo(T o) b) compare(T a, T b) c) equals(T o) d) hashCode()  
    **Answer:** b
32. Using Comparator avoids modifying the class:  
    a) True b) False  
    **Answer:** a
33. Collections.sort(list, comp) requires:  
    a) list elements implement Comparable b) comp implements Comparator c) list is Set d) no nulls  
    **Answer:** b
34. Comparator.comparing() returns a:  
    a) Comparable b) Comparator c) Function d) Collector  
    **Answer:** b
35. Chaining comparators using .thenComparing(...) allows:  
    a) Multi-level sorting b) Reversing order c) Filtering duplicates d) Ignoring nulls  
    **Answer:** a
36. To sort descending you can use:  
    a) comp.reversed() b) comp.negate() c) comp.invert() d) comp.backward()  
    **Answer:** a
37. A Comparator must be:  
    a) Serializable b) Cloneable c) Stateless (ideally) d) Iterable  
    **Answer:** c
38. A comparator that is inconsistent with equals may:  
    a) Cause strange Set behavior b) Compile error c) Better performance d) UnsupportedOperationException  
    **Answer:** a
39. TreeSet<>(comp) uses comp to:  
    a) Check nulls b) Compare elements for ordering c) Manage hash buckets d) Handle serialization  
    **Answer:** b
40. To compare strings ignoring case, you’d use:  
    a) Comparable<String> b) String.CASE\_INSENSITIVE\_ORDER c) Collections.reverseOrder() d) Collator  
    **Answer:** b
41. Which will fail fast on concurrent modification?  
    a) LinkedList iterator b) HashSet iterator c) ArrayList iterator d) All of the above  
    **Answer:** d
42. Calling TreeSet.first() on an empty TreeSet:  
    a) Returns null b) Throws NoSuchElementException c) Returns default value d) Infinite loop  
    **Answer:** b
43. HashSet initial capacity defaults to:  
    a) 8 b) 16 c) 32 d) 64  
    **Answer:** b
44. HashSet.contains(element) average complexity:  
    a) O(1) b) O(n) c) O(log n) d) O(n log n)  
    **Answer:** a
45. LinkedList implements:  
    a) List, Queue, Deque b) List only c) Deque only d) List, Set  
    **Answer:** a
46. You can convert ArrayList to an Array using:  
    a) list.toArr() b) list.toArray() c) list.copyOf() d) list.asList()  
    **Answer:** b
47. Which code sorts strings in reverse alphabetical using Comparator?  
    a) sort(list, Comparator.reverseOrder())  
    b) sort(list, Comparator.naturalOrder().reversed())  
    c) Both a and b  
    d) Neither  
    **Answer:** c
48. Which Set isn't ordered?  
    a) TreeSet b) LinkedHashSet c) HashSet d) SortedSet  
    **Answer:** c
49. **Comparator.nullsFirst(Comparator.naturalOrder())** ensures:  
    a) Nulls thrown exception b) Nulls never appear c) Null elements sorted first d) Nulls at end  
    **Answer:** c

**EXAMPLE OF MULTIPLE SORTING:**

import java.util.\*;  
class Person{  
  private String name;  
  private int age;  
  
  public String getName() {  
    return name;  
  }

  public int getAge() {  
    return age;  
  }  
  
  
  public Person(String name,int age){  
    this.name=name;  
    this.age=age;  
  }  
  
  public String toString(){  
    return this.name + " " + this.age;  
  }  
}

class PersonMain{  
  public static void main(String[] args){  
    List<Person> person = Arrays.asList(  
      new Person("Brian",30),  
      new Person("Harry",23),  
      new Person("Brian",25)  
    );  
  
    System.out.println("Original List");  
    System.out.println(person);  
  
    person.sort(Comparator.comparing((Person p)->p.getName()).thenComparing(p->p.getAge()));  
  
    System.out.println("Sorted list by name then by age: ");  
    for(Person p:person){  
      System.out.println(p);  
    }    
  
  }  
}

**Map Interface**

The Map interface is part of the Java Collections Framework and represents a data structure that maps keys to values. It is not a class but an interface that several classes like HashMap, TreeMap, and LinkedHashMap implement.

**Map Interface Overview**

**Key Characteristics:**

* A map **cannot contain duplicate keys**.
* Each key maps to **exactly one value**.
* A key can map to null, and a Map can contain multiple null values, but only **one null key** (in HashMap).

**Common Implementations:**

| **Class** | **Description** |
| --- | --- |
| **HashMap** | **Unordered, allows one null key and many null values** |
| **TreeMap** | **Sorted by natural order or a custom Comparator; doesn’t allow null keys** |
|  |  |

**Common Methods in Map<K, V>:**

V put(K key, V value);           // Adds or replaces a key-value pair

V get(Object key);                // Returns value for the key

V remove(Object key);            // Removes the key-value pair

boolean containsKey(Object key); // Checks if key exists

boolean containsValue(Object value);

Set<K> keySet();                 // Returns set of keys

Collection<V> values();         // Returns collection of values

Set<Map.Entry<K, V>> entrySet(); // Returns set of key-value pairs

void clear();                   // Removes all entries

int size();                     // Number of entries

boolean isEmpty();              // Checks if empty

**Example**

import java.util.\*;

public class MapExample {

   public static void main(String[] args) {

       Map<String, Integer> map = new HashMap<>();

      map.put("Apple", 10);

      map.put("Banana", 20);

      map.put("Orange", 30);

      System.out.println("Value for 'Apple': " + map.get("Apple"));

       for (Map.Entry<String, Integer> entry : map.entrySet()) {

          System.out.println(entry.getKey() + " => " + entry.getValue());

       }

      map.remove("Banana");

      System.out.println("After removal: " + map);

   }

}

**When to Use Which Map**

* HashMap: Best general-purpose map.
* TreeMap: When you need sorted keys.

**HashMap in Java**

HashMap is a class in Java that implements the **Map interface** and **uses a hash table** for storing key-value pairs.

**Key Features of HashMap:**

* **Stores key-value pairs**
* **Allows one null key** and **multiple null values**
* **Does not maintain order**
* **Non-synchronized** (not thread-safe by default)
* **Offers constant-time performance** (O(1)) for basic operations: put, get, remove

**Syntax:**

import java.util.HashMap;

HashMap<KeyType, ValueType> map = new HashMap<>();

**Example:**

HashMap<String, Integer> map = new HashMap<>();

**Common Methods:**

| **Method** | **Description** |
| --- | --- |
| put(K key, V value) | Adds or updates a key-value pair |
| get(Object key) | Returns the value for the given key |
| remove(Object key) | Removes the entry with the given key |
| containsKey(Object key) | Checks if a key exists |
| containsValue(Object value) | Checks if a value exists |
| size() | Returns the number of entries |
| clear() | Removes all entries |
| isEmpty() | Checks if the map is empty |
| keySet() | Returns a set of keys |
| values() | Returns a collection of values |
| entrySet() | Returns a set of key-value pairs (Map.Entry) |

**Example: HashMap**

import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

   public static void main(String[] args) {

      HashMap<String, Integer> map = new HashMap<>();

      map.put("Apple", 50);

      map.put("Banana", 20);

      map.put("Orange", 30);

      map.put("Banana", 25); // Overwrites old value

      System.out.println("Apple Price: " + map.get("Apple"));

       for (Map.Entry<String, Integer> entry : map.entrySet()) {

          System.out.println(entry.getKey() + " => " + entry.getValue());

       }

      map.remove("Orange");

      System.out.println("Map after removal: " + map);

   }

}

**Internal Working of HashMap**

* Uses **buckets** internally, where each bucket is a LinkedList or a TreeNode (after Java 8, for high-collision buckets).
* A key's hashCode() determines the bucket index.

**Important Notes**

* Not thread-safe
* Always override equals() and hashCode() in custom key classes.

**TreeMap in Java**

A TreeMap is a part of Java’s java.util package, and it implements the Map interface. **Unlike HashMap, it stores keys in a sorted (natural or custom) order using a Red-Black Tree,** which is a self-balancing binary search tree.

**Features of TreeMap:**

* Stores **key-value pairs** sorted by **keys**
* **No null keys** allowed (throws NullPointerException)
* Allows **multiple null values**
* Implements NavigableMap and SortedMap interfaces
* Not synchronized (not thread-safe by default)
* Logarithmic time complexity: **O(log n)** for get, put, remove

**Declaration:**

import java.util.TreeMap;

TreeMap<KeyType, ValueType> map = new TreeMap<>();

**Example:**

TreeMap<String, Integer> map = new TreeMap<>();

**Common Methods:**

| **Method** | **Description** |
| --- | --- |
| put(K key, V value) | Adds or updates a key-value pair |
| get(Object key) | Retrieves value associated with key |
| remove(Object key) | Removes entry by key |
| firstKey() / lastKey() | Returns first/last key in sorted order |
| higherKey(K key) | Returns next greater key |
| lowerKey(K key) | Returns previous lesser key |
| keySet() | Returns keys in ascending order |
| entrySet() | Returns sorted set of key-value pairs |
| headMap(K toKey) | Returns map with keys less than toKey |
| tailMap(K fromKey) | Returns map with keys greater than or equal to fromKey |
| subMap(K fromKey, K toKey) | Returns map within the key range |

**Example: Basic TreeMap**

import java.util.TreeMap;

import java.util.Map;

public class TreeMapExample {

   public static void main(String[] args) {

      TreeMap<String, Integer> map = new TreeMap<>();

      map.put("Banana", 20);

      map.put("Apple", 50);

      map.put("Orange", 30);

      System.out.println("Sorted Map:");

       for (Map.Entry<String, Integer> entry : map.entrySet()) {

          System.out.println(entry.getKey() + " => " + entry.getValue());

       }

      System.out.println("First Key: " + map.firstKey());

      System.out.println("Last Key: " + map.lastKey());

      System.out.println("Higher Key than 'Apple': " + map.higherKey("Apple"));

   }

}

**Example with Custom Comparator (Reverse Order):**

import java.util.\*;

public class TreeMapCustomSort {

   public static void main(String[] args) {

      TreeMap<String, Integer> map = new TreeMap<>(Collections.reverseOrder());

      map.put("Banana", 20);

      map.put("Apple", 50);

      map.put("Orange", 30);

      System.out.println("Reverse Sorted Map:");

       for (Map.Entry<String, Integer> entry : map.entrySet()) {

          System.out.println(entry.getKey() + " => " + entry.getValue());

       }

   }

}

**TreeMap vs HashMap**

| **Feature** | **HashMap** | **TreeMap** |
| --- | --- | --- |
| Order | Unordered | Sorted by key |
| Null keys | Allows one null key | No null keys |
| Performance | O(1) for get/put | O(log n) for get/put |
| Thread-safe | No | No |
| Backed by | Hash table | Red-Black Tree |

import java.util.TreeMap;  
import java.util.Map;  
class TreeMapExample {  
    public static void main(String[] args) {  
        TreeMap<String, Integer> map = new TreeMap<>();

        map.put("Banana", 20);  
        map.put("Apple", 50);  
        map.put("Orange", 30);

        System.out.println("Sorted Map:");  
        for (Map.Entry<String, Integer> entry : map.entrySet()) {  
            System.out.println(entry.getKey() + " => " + entry.getValue());  
        }

        System.out.println("First Key: " + map.firstKey());  
        System.out.println("Last Key: " + map.lastKey());  
        System.out.println("Higher Key than 'Apple': " + map.higherKey("Apple"));  
    }  
}

import java.util.TreeMap;  
import java.util.Map;  
import java.util.Collections;  
class TreeMapExample {  
    public static void main(String[] args) {  
       TreeMap<String, Integer> map = new TreeMap<>(Collections.reverseOrder());

        map.put("Banana", 20);  
        map.put("Apple", 50);  
        map.put("Orange", 30);

        System.out.println("Reverse Sorted Map:");  
        for (Map.Entry<String, Integer> entry : map.entrySet()) {  
            System.out.println(entry.getKey() + " => " + entry.getValue());  
        }

        System.out.println("First Key: " + map.firstKey());  
        System.out.println("Last Key: " + map.lastKey());  
        System.out.println("Higher Key than 'Apple': " + map.higherKey("Apple"));  
    }  
}

**Input/Output operations in Java**

**What is Java I/O?**

Java I/O (Input and Output) is used to process **input** and produce **output**.

***Java provides the java.io and java.nio packages for these operations.***

*In* ***java.nio******“n”*** *understands for* ***New***

**Two Main Types of I/O**

1. **Byte Streams** (InputStream / OutputStream)
   * Used for reading and writing **binary data** (images, audio, etc.)
   * Base classes:
     + **InputStream:** For reading bytes
     + **OutputStream:**For writing bytes
2. **Character Streams** (Reader / Writer)
   * Used for reading and writing **text data**
   * Base classes:
     + **Reader:** For reading characters
     + **Writer:** For writing characters

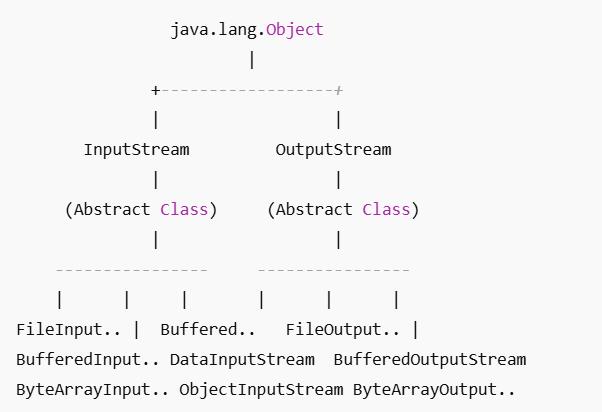
**Commonly Used I/O Classes**

| **Type** | **Class Name** | **Description** |
| --- | --- | --- |
| Byte Input | FileInputStream | Reads data from a file |
| Byte Output | FileOutputStream | Writes data to a file |
| Character Input | FileReader | Reads characters from a file |
| Character Output | FileWriter | Writes characters to a file |
| Buffered Input | BufferedReader | Reads text efficiently, line by line |
| Buffered Output | BufferedWriter | Writes text efficiently |
| Print Output | PrintWriter | Writes formatted text to a file |

                                                                          -------------

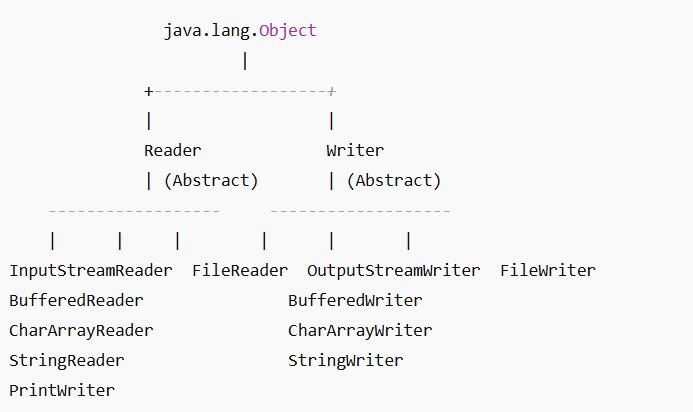
**Java I/O Class Hierarchy (from java.io package)**

**1. Byte Streams**



* **Common Byte Stream Classes:**
  + FileInputStream / FileOutputStream
  + BufferedInputStream / BufferedOutputStream
  + DataInputStream / DataOutputStream
  + ObjectInputStream / ObjectOutputStream
  + ByteArrayInputStream / ByteArrayOutputStream

**2. Character Streams**



* **Common Character Stream Classes:**
  + FileReader / FileWriter
  + BufferedReader / BufferedWriter
  + InputStreamReader / OutputStreamWriter
  + PrintWriter, CharArrayReader, StringWriter

File Handling

**File handling** in Java allows you to create, read, write, update, and delete files. It is primarily done using classes in the java.io and java.nio.file packages.

**Commonly Used Classes**

| **Class** | **Purpose** |
| --- | --- |
| File | Represents file/directory path |
| FileReader / FileWriter | For reading/writing text files |
| BufferedReader / BufferedWriter | For efficient reading/writing |
| FileInputStream / FileOutputStream | For reading/writing binary files |
| PrintWriter | For writing formatted text |
| Files (from java.nio.file) | For advanced file operations |

**Example 1: Create a File**

import java.io.File;

import java.io.IOException;

public class CreateFileExample {

   public static void main(String[] args) {

       File file = new File("myfile.txt");

       try {

           if (file.createNewFile()) {

              System.out.println("File created: " + file.getName());

           } else {

              System.out.println("File already exists.");

           }

       } catch (IOException e) {

           e.printStackTrace();

       }

   }

}

**Example 2: Write to a File**

import java.io.FileWriter;

import java.io.IOException;

public class WriteFileExample {

   public static void main(String[] args) {

       try (FileWriter writer = new FileWriter("myfile.txt")) {

          writer.write("Hello, Java File Handling!");

       } catch (IOException e) {

          e.printStackTrace();

       }

   }

}

**Example 3: Read from a File**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFile {

   public static void main(String[] args) {

              try (BufferedReader reader = new BufferedReader(new FileReader("myfile.txt"))) {

           String line;

           while ((line = reader.readLine()) != null) {

              System.out.println(line);

           }

       } catch (IOException e) {

          e.printStackTrace();

       }

   }

}

**Example 4: Delete a File**

import java.io.File;

public class DeleteFile {

   public static void main(String[] args) {

       File file = new File("myfile.txt");

       if (file.delete()) {

          System.out.println("Deleted the file: " + file.getName());

       } else {

          System.out.println("Failed to delete the file.");

       }

file.close();

}

}

**File Handling Tips**

* Always **close** streams (FileReader, FileWriter, etc.) to avoid memory leaks.
* Use **try-with-resources** (Java 7+) to auto-close resources.
* Check file **existence** with **file.exists()** before operations.

**Serialization in Java**

**Serialization is the process of converting a Java object into a byte stream** so that it can be saved to a file, sent over a network, or stored in memory for later reconstruction (deserialization).

**Why Use Serialization?**

* Save object state permanently (to file, database, etc.)
* Transfer objects over network (e.g., in RMI or sockets)
* Deep copy objects

**Core Classes & Interface**

| **Element** | **Description** |
| --- | --- |
| Serializable | Marker interface to make a class serializable |
| ObjectOutputStream | Writes objects to an OutputStream |
| ObjectInputStream | Reads objects from an InputStream |

***NOTE: A class must implement Serializable to be eligible for serialization.***

**Example**

**Step 1: Create a Serializable Class**

import java.io.Serializable;

public class Student **implements Serializable** {

   private String name;

   private int age;

   public Student(String name, int age) {

       this.name = name;

       this.age = age;

   }

   public void display() {

      System.out.println("Name: " + name + ", Age: " + age);

   }

}

**Step 2: Serialize the Object (Write to File)**

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

public class SerializeExample {

   public static void main(String[] args) {

       Student s1 = new Student("Brian",20);

       try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.ser"))) {

          out.writeObject(s1);

          System.out.println("Object serialized to student.ser");

       } catch (Exception e) {

          e.printStackTrace();

       }

   }

}

**Step 3: Deserialize the Object (Read from File)**

import java.io.FileInputStream;

import java.io.ObjectInputStream;

public class DeserializeExample {

   public static void main(String[] args) {

       try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"))) {

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

          s.display();

       } catch (Exception e) {

           e.printStackTrace();

       }

   }

}

**Points to remember:**

* transient keyword: Fields marked as transient are **not serialized**.
* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Serialization does **not store static fields**.
* Java uses **default serialization**, but you can customize it with:

**Concept**

**transient Keyword**

Fields marked transient are **not serialized**.

transient String password; // This won't be saved during serialization

**serialVersionUID**

A unique identifier to ensure version compatibility during deserialization.

private static final long serialVersionUID = 1L;

If not defined, JVM generates one. If the class structure changes, deserialization may fail with InvalidClassException.

**Custom Serialization**

To control serialization manually, define these methods:

private void writeObject(ObjectOutputStream oos) throws IOException {

   // Custom serialization logic

}

private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {

   // Custom deserialization logic

}

**serialVersionUID=1L**

**What is serialVersionUID?**

* It's a **version control ID** for serialized classes.
* If a class is **serialized and later modified**, and the serialVersionUID doesn't match during deserialization, the JVM will throw:

***java.io.InvalidClassException***

**Example Without serialVersionUID**

**Step 1: Original Version of the Class**

import java.io.Serializable;

**public class User implements Serializable {**

//private static final long serialVersionUID = 1L;

   String username;

   String email;

Int age;

   public User(String username, String email) {

       this.username = username;

       this.email = email;

   }

}

**Serialize This Object**

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

public class SerializeUser {

   public static void main(String[] args) throws Exception {

       User user = new User("alice", "alice@example.com");

      ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("user.ser"));

      out.writeObject(user);

       out.close();

   }

}

**This will create user.ser.**

**Step 2: Modify the Class (Structure Change)**

Now imagine you modify the User class by **adding a new field**:

import java.io.Serializable;

public class User implements Serializable {

   String username;

   String email;

   int age;  // New field has bee added

}

**Try to Deserialize the Old File**

import java.io.FileInputStream;

import java.io.ObjectInputStream;

public class DeserializeUser {

   public static void main(String[] args) throws Exception {

      ObjectInputStream in = new ObjectInputStream(new FileInputStream("user.ser"));

       User user = (User) in.readObject();  // This will fail

       in.close();

   }

}

**This Error:**

java.io.InvalidClassException: User; local class incompatible:

stream classdesc serialVersionUID = [old], local class serialVersionUID = [new]

**Solution: Add serialVersionUID**

To maintain compatibility between versions, **manually define** a serialVersionUID:

**private static final long serialVersionUID = 1L;**

If both versions of the class (before and after modification) have the **same serialVersionUID**, deserialization will **not fail**, even if structure has changed — though you must handle missing fields (e.g., age will default to 0 if not present in the stream).

**Summary**

| **Situation** | **Result** |
| --- | --- |
| Class changes, no serialVersionUID | InvalidClassException |
| Class changes, same serialVersionUID | Deserialization succeeds (with default values for new fields) |
| Class unchanged | Always safe |

**Functional Interfaces**

1. **Predicate<T>**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Tests a condition on an input and returns a boolean result |
| **Method** | boolean test(T t) |
| **Input** | Yes (1 input of type T) |
| **Output** | Boolean |
| **Use Case** | Filtering, validations, conditions |

**Example:**

Predicate<String> isLong = s -> s.length() > 5;

System.out.println(isLong.test("hello")); // false

**2. Function<T, R>**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Accepts one argument and returns a result |
| **Method** | R apply(T t) |
| **Input** | Yes (1 input of type T) |
| **Output** | Yes (1 output of type R) |
| **Use Case** | Data transformation, mapping |

**Example:**

Function<String, Integer> lengthFunc = s -> s.length();

System.out.println(lengthFunc.apply("Hello")); // 7

**3. Runnable**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Represents a task to be run (no input/output) |
| **Method** | void run() |
| **Input** | No |
| **Output** | No |
| **Use Case** | Threads, background tasks |

**Example:**

Runnable task = () -> System.out.println("Running...");

task.run(); // Output: Running...

**4. Supplier<T>**

| **Feature** | **Description** |
| --- | --- |
| **Purpose** | Supplies a result (no input) |
| **Method** | T get() |
| **Input** | No |
| **Output** | Yes (value of type T) |
| **Use Case** | Lazy loading, generating values |

**Example:**

Supplier<String> message = () -> "Hello!";

System.out.println(message.get()); // Hello!

**Summary Table**

| **Interface** | **Input** | **Output** | **Return Type** | **Main Method** | **Use Case** |
| --- | --- | --- | --- | --- | --- |
| **Predicate** | Yes | Yes | Boolean | test(T t) | Filtering/validation |
| **Function** | Yes | Yes | Any type R | apply(T t) | Mapping/transformation |
| **Runnable** | No | No | Void | run() | Tasks/threads |
| **Supplier** | No | Yes | Any type T | get() | Value generation |

**MCQ – Lambda Expression**

1. What will be the output of the following code?

List<String> list = Arrays.asList("a", "bb", "ccc");

list.forEach(s -> System.out.print(s.length() + " "));

A) 1 2 3  
B) a bb ccc  
C) 3 2 1  
D) Compile error  
Answer: A

2. Choose the correct lambda that squares an integer.

A) (x) -> x  
B) (x) -> x \* x  
C) (x) -> x + x  
D) x -> x \* 2  
Answer: B

3. Which lambda expression correctly sorts a list of strings in reverse order?

List<String> list = Arrays.asList("a", "b", "c");

A) list.sort((a, b) -> a.compareTo(b));  
B) list.sort((a, b) -> b.compareTo(a));  
C) list.sort((a, b) -> a.length() - b.length());  
D) list.sort((a, b) -> a + b);  
Answer: B

4. What is the output?

Runnable r = () -> System.out.print("Hello");

r.run();

A) No output  
B) Hello  
C) Compile error  
D) Runtime error  
Answer: B

\*\*\*\*\*5. Which of these will compile?

A) Predicate<String> p = (s) -> s.isEmpty();  
B) Function<Integer> f = (i) -> i \* 2;  
C) Consumer<String> c = (s) -> return s;  
D) Supplier<String> s = () -> System.out.println("Hello");  
Answer: A

6. Which functional interface is best for a lambda that takes two integers and returns their sum?

A) Predicate<Integer>  
**B) BiFunction<Integer, Integer, Integer>**  
C) Supplier<Integer>  
D) Consumer<Integer>  
Answer: B

7. What is the output?

List<Integer> nums = Arrays.asList(1, 2, 3);

nums.stream().map(n -> n \* 2).forEach(System.out::print);

A) 123  
B) 246  
C) 135  
D) 6  
Answer: B

8. Choose the correct lambda for checking if a string starts with 'A':

A) s -> s.endsWith("A")  
B) s -> s.startsWith("A")  
C) (String s) -> s.contains("A")  
D) () -> s.startsWith("A")  
Answer: B

9. Output of the code?

Predicate<String> p = s -> s.length() > 3;

System.out.print(p.test("Hi"));

A) true  
B) false  
C) Compile error  
D) Runtime error  
Answer: B

10. Which lambda returns a constant value 10?

A) () -> 10  
B) (x) -> x  
C) (x) -> 10 \* x  
D) (x) -> x + 10  
Answer: A

11. Output of this code:

Stream.of("A", "B", "C").forEach(ch -> System.out.print(ch + " "));

A) A B C  
B) ABC  
C) Compile error  
D) A, B, C  
Answer: A

12. What is the return type of the lambda below?

Function<String, Integer> f = s -> s.length();

A) String  
B) int  
C) Integer  
D) boolean  
Answer: C

13. Output of the following code:

Function<Integer, Integer> doubleIt = x -> x \* 2;

System.out.print(doubleIt.apply(5));

A) 10  
B) 25  
C) 5  
D) Compile error  
Answer: A

14. Which of these is a valid Consumer lambda?

A) x -> x \* 2  
B) x -> return x;  
C) x -> System.out.println(x)  
D) () -> x + 2  
Answer: C

15. Which lambda is valid for Supplier<String>?

A) () -> "Hello"  
B) (s) -> "Hello"  
C) "Hello" -> ()  
D) s -> return "Hello"  
Answer: A

16. What does this lambda do?

Function<String, String> f = s -> s.toUpperCase();

A) Returns lowercase  
B) Returns uppercase  
C) Returns first character  
D) Returns length  
Answer: B

17. Output of this code?

Predicate<Integer> isPositive = x -> x > 0;

System.out.print(isPositive.test(-1));

A) true  
B) false  
C) -1  
D) Compile error  
Answer: B

18. What is true about this code?

List<String> list = Arrays.asList("a", "ab", "abc");

list.removeIf(s -> s.length() < 2);

A) Removes strings of length < 2  
B) Adds strings of length < 2  
C) Sorts strings  
D) Filters nothing  
Answer: A

19. Choose valid lambda for BiPredicate<String, String>:

A) (s1, s2) -> s1.equals(s2)  
B) (s) -> s.isEmpty()  
C) () -> true  
D) (x, y, z) -> true  
Answer: A

20. What is output?

IntStream.range(1, 4).forEach(i -> System.out.print(i));

A) 123  
B) 012  
C) 134  
D) 234  
Answer: A