JAVA

What is Java? Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It follows the principle of "Write Once, Run Anywhere" (WORA), meaning compiled Java code can run on all platforms that support Java without needing recompilation.

Advantages and Disadvantages of Java

Advantages	Disadvantages
Platform-independent (WORA)	Slower compared to languages like C++
Rich API	High memory consumption
Secure and robust	Verbose syntax
Automatic memory management	GUI development is complex
Multithreading support	Limited support for low-level programming

---

JVM, JRE, and JDK

- **JVM (Java Virtual Machine)**: It is an abstract machine that enables a computer to run Java programs.
- **JRE (Java Runtime Environment)**: It includes libraries and JVM required for running Java applications.
- **JDK (Java Development Kit)**: It includes JRE along with development tools like the compiler and debugger.

Difference between JVM, JRE, and JDK

Feature	JVM	JRE	JDK
Contains JVM	Yes	Yes	Yes
Contains JRE	No	Yes	Yes

Feature	JVM	JRE	JDK
Contains development tools	No	No	Yes
Purpose	Executes Java bytecode	Runs Java applications	Develops and runs Java programs

---

Basic Java Syntax

Example of a Java Program:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Anupam");
    }
}
```

```
Hello, Anupam
```

Variables in Java:

int a = 10; // Integer variable

float b = 5.5f; // Floating-point variable

char c = 'A'; // Character variable

String name = "Java"; // String variable

boolean b = true; // Boolean variable

double variable is also there and so on.

Operators in Java:

int sum = 10 + 20; // Arithmetic Operator

boolean isJava = true && false; // Logical Operator

int result = (a > b) ? a : b; // Ternary Operator

Control Structures:

If-Else Example:

```
if (a > b) {
   System.out.println("A is greater");
} else {
   System.out.println("B is greater");
}
```

Loop Structures:

For Loop:

```
for (int i = 0; i < 5; i++) {
   System.out.println(i);
}
```

While Loop:

```
int i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
```

Do-While Loop:

```
int i = 0;
do {
   System.out.println(i);
   i++;
} while (i < 5);
```

Switch Statement:

```
int day = 3;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  default:
    System.out.println("Other day");
}
```

Creating an Array in Java

Different Ways to Create an Array:

```
int[] arr1 = new int[5]; // Declaration and allocation
int[] arr2 = {1, 2, 3, 4, 5}; // Declaration and initialization
```

Jagged Array in Java

```
int[][] jaggedArr = new int[3][];
jaggedArr[0] = new int[2];
jaggedArr[1] = new int[3];
jaggedArr[2] = new int[1];
```

Enhanced For Loop in Java

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
  System.out.println(num);
```

}

Final Keyword in Java

- **Final Variable:** Its value cannot be changed after initialization.
- **Final Method:** Cannot be overridden by subclasses.
- **Final Class:** Cannot be inherited.

```
final class Constants {
   final int VALUE = 100;
}
```

Memory Management in Java Java memory is divided into Stack and Heap.

- **Stack Memory:** Stores method-specific values and local variables.
- **Heap Memory:** Stores objects and class instances.

How to Increase Heap Size: Use JVM arguments:

```
java -Xms512m -Xmx1024m MyClass
```

Example: Setting Heap Size to 512MB (Initial) and 2GB (Maximum)

```sh
java -Xms512m -Xmx2g MyApplication
```

Wrapper Classes Wrapper classes convert primitive types into objects.

```
Integer obj = Integer.valueOf(10);
int num = obj.intValue();
```

Object-Oriented Programming (OOP) in Java OOP is a programming paradigm based on objects that contain data and methods.

Pillars of OOP:

1. **Encapsulation** - Hiding data using private access.
2. **Inheritance** - Acquiring properties of parent class.
3. **Polymorphism** – Message displayed in more than one form(Method overloading and overriding).
4. **Abstraction** - Hiding implementation details.

Static Method and Static Block in Java

Static Method

A **static method** in Java is a method that belongs to the class rather than instances (objects) of the class. It can be called without creating an object of the class.

Characteristics of Static Methods:

1. Declared using the static keyword.
2. Belongs to the class, not any specific instance.
3. Can be called using the class name.
4. Cannot access non-static (instance) variables or methods directly.
5. Commonly used for utility functions or operations that do not require object-specific data.

Example of a Static Method

```
class Utility {
   static void displayMessage() {
      System.out.println("Static Method Called");
   }
}
```

```
public class Main {
  public static void main(String[] args) {
    // Calling static method using class name
    Utility.displayMessage();
  }
}
```

Output:

Static Method Called

Static Block

A **static block** is a block of code that executes **only once** when the class is loaded into memory. It is mainly used for **initialization** of static variables.

Characteristics of Static Block:

1. Declared using the static keyword.
2. Executes **automatically** when the class is loaded, even before the main method.
3. Can initialize static variables.
4. Can be used for logging or one-time setup tasks.

Example of a Static Block

```
class Example {
  static {
    System.out.println("Static Block Executed");
  }

  public static void main(String[] args) {
    System.out.println("Main Method Executed");
  }
```

}

Output:

Static Block Executed

Main Method Executed

✅ **Notice**: The static block runs **before** the main method.

Difference Between Static Method and Static Block

Feature	**Static Method**	**Static Block**
Purpose	Performs specific actions (like utility functions)	Used to initialize static variables or run code once
Execution	Called explicitly using class name	Runs automatically when the class loads
When it runs	When invoked	Before main() when the class is loaded
Can access instance variables?	❌ No	❌ No
Example Use Case	Utility functions like Math.pow()	Setting up database connections, initializing static variables

---

Example of Static Block and Static Method Together

```
class Test {
   static int a;
  // Static block executes first
   static {
      a = 10;
      System.out.println("Static Block Initialized, a = " + a);
```

```
 }
  // Static method
  static void display() {
     System.out.println("Static Method Called, a = " + a);
  }

  public static void main(String[] args) {
     System.out.println("Main Method Executed");
     display(); // Calling static method
  }
}
```

Output:

Static Block Initialized, a = 10

Main Method Executed

Static Method Called, a = 10

✅ **Key Takeaways**:

- **Static block runs first** when the class is loaded.
- **Static method runs only when explicitly called**.

Wrapper Class in Java

A **Wrapper Class** is a class that encapsulates (wraps) primitive data types into objects. Java provides wrapper classes for all primitive data types to allow object manipulation.

Why Use Wrapper Classes?

1. **Collection Framework Compatibility** – Collections like ArrayList work with objects, not primitives.
2. **Utility Methods** – Wrapper classes provide useful methods for type conversions.

3. **Autoboxing & Unboxing** – Automatic conversion between primitives and wrapper objects.

Wrapper Classes in Java

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Example of Wrapper Class Usage

```
public class WrapperExample {
    public static void main(String[] args) {
        // Autoboxing (Primitive to Object)
        Integer num = 10;
        Double decimal = 20.5;

        // Unboxing (Object to Primitive)
        int n = num;
        double d = decimal;

        System.out.println("Integer Object: " + num);
        System.out.println("Primitive int: " + n);
```

```
    }
}
```

Output:

Integer Object: 10

Primitive int: 10

Upcasting and Downcasting in Java

Upcasting

- **Converting a subclass object to a superclass reference.**
- **Happens implicitly** and allows access to only superclass members.

Example of Upcasting

```
class Parent {
    void show() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child class method");
    }
}

public class UpcastingExample {
    public static void main(String[] args) {
        Parent obj = new Child(); // Upcasting
```

```
        obj.show(); // Allowed
        // obj.display(); // Not allowed (Parent reference can't access child-specific methods)
    }
}
```

Output:

Parent class method

Downcasting

- **Converting a superclass reference back to a subclass object.**
- **Needs explicit casting** and can throw ClassCastException if not done correctly.

Example of Downcasting

```
class Parent {
    void show() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child class method");
    }
}

public class DowncastingExample {
    public static void main(String[] args) {
        Parent parent = new Child(); // Upcasting
```

```
        Child child = (Child) parent; // Downcasting
        child.display(); // Now accessible
    }
}
```

Output:

Child class method

⚠ **If we try to downcast an unrelated object, we get ClassCastException.**

```
Parent p = new Parent();
Child c = (Child) p; // This will throw ClassCastException at runtime
```

Key Differences Between Upcasting and Downcasting

Feature	**Upcasting**	**Downcasting**
Direction	Subclass → Superclass	Superclass → Subclass
Type of Casting	Implicit	Explicit
Access to Members	Only superclass members	Both superclass and subclass members
Risk of ClassCastException	No	Yes, if done incorrectly

Object in Java

An **object** is an instance of a class that has **state (fields/variables)** and **behavior (methods)**. Objects are created from classes and are stored in **heap memory**.

Example of an Object in Java

```
class Car {
    String brand;
    int speed;
```

```
    void show() {
        System.out.println("Car Brand: " + brand + ", Speed: " + speed);
    }
}

public class ObjectExample {
    public static void main(String[] args) {
        Car myCar = new Car();  // Creating an object
        myCar.brand = "Tesla";
        myCar.speed = 120;
        myCar.show();
    }
}
```

Output:

Car Brand: Tesla, Speed: 120

Class in Java

A **class** is a blueprint or template for creating objects. It defines **fields (variables)** and **methods (functions)** that objects will have.

Syntax of a Class

```
class ClassName {
    // Fields (variables)
    // Methods (functions)
}
```

Example of a Class in Java

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

   void display() {
      System.out.println("Name: " + name + ", Age: " + age);
   }
}

public class ClassExample {
   public static void main(String[] args) {
      Student s1 = new Student();
      s1.name = "John";
      s1.age = 20;
      s1.display();
   }
}
```

Output:

Name: John, Age: 20

Inheritance in Java

Definition: Inheritance is a mechanism in Java where one class **inherits** the properties and methods of another class. It promotes **code reusability**.

Types of Inheritance in Java

Type	Description
Single Inheritance	One class inherits from another.
Multilevel Inheritance	A class inherits from another class, which in turn inherits from another class.
Hierarchical Inheritance	Multiple classes inherit from a single class.
Multiple Inheritance (via Interfaces)- Not possible in java	A class implements multiple interfaces.

```java
// 1. Single Inheritance Example
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

public class SingleInheritance {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat(); // Inherited method
        d.bark(); // Child class method
    }
}
```

```java
// 2. Multilevel Inheritance Example
class Grandparent {
    void grandparentMethod() {
        System.out.println("Grandparent method");
    }
}

class Parent extends Grandparent {
    void parentMethod() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void childMethod() {
        System.out.println("Child method");
    }
}

public class MultilevelInheritance {
    public static void main(String[] args) {
        Child c = new Child();
        c.grandparentMethod(); // Inherited from Grandparent
        c.parentMethod(); // Inherited from Parent
        c.childMethod(); // Own method
    }
}
```

```java
// 3. Hierarchical Inheritance Example
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {
    void fourWheeler() {
        System.out.println("Car has four wheels");
    }
}

class Bike extends Vehicle {
    void twoWheeler() {
        System.out.println("Bike has two wheels");
    }
}

public class HierarchicalInheritance {
    public static void main(String[] args) {
        Car car = new Car();
        Bike bike = new Bike();

        car.run();
        car.fourWheeler();

        bike.run();
        bike.twoWheeler();
    }
}
```

```java
// 4. Multiple Inheritance (via Interfaces) Example
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class MultipleInheritanceExample implements A, B {
    public void methodA() {
        System.out.println("Method A from Interface A");
    }

    public void methodB() {
        System.out.println("Method B from Interface B");
    }

    public static void main(String[] args) {
        MultipleInheritanceExample obj = new MultipleInheritanceExample();
        obj.methodA();
        obj.methodB();
    }
}
```

Encapsulation in Java

Definition: Encapsulation is the process of wrapping **data (variables)** and **methods** within a single unit (**class**) and restricting direct access to some of the object's details.

Example of Encapsulation

```
class Employee {

  private String name;  // Private variable


  // Getter method

  public String getName() {

    return name;

  }
```

```
    // Setter method
    public void setName(String newName) {
        name = newName;
    }
}

public class EncapsulationExample {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.setName("John Doe");
        System.out.println("Employee Name: " + emp.getName());
    }
}
```

Output:

Employee Name: John Doe

---

20) Getter and Setter Methods

Getter and Setter methods allow controlled access to private variables.

Example of Getters and Setters

```
class Person {
    private int age;

    // Getter
    public int getAge() {
        return age;
    }
```

```
    // Setter
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive!");
        }
    }
}

public class GetterSetterExample {
    public static void main(String[] args) {
        Person p = new Person();
        p.setAge(25);
        System.out.println("Age: " + p.getAge());
    }
}
```

---

21) Access Modifiers in Java

Access Modifiers define the **scope (visibility)** of variables, methods, and classes.

Types of Access Modifiers

Modifier	Scope	Accessible in Same Class	Accessible in Same Package	Accessible in Subclass	Accessible from Outside
private	Within the same class	✅	❌	❌	❌
default (no keyword)	Within the same package	✅	✅	❌	❌
protected	Within the same package and subclasses	✅	✅	✅	❌
public	Anywhere	✅	✅	✅	✅

Example of Access Modifiers

```
class AccessExample {
    private int privateVar = 10;
    public int publicVar = 20;
    protected int protectedVar = 30;
    int defaultVar = 40; // Default access

    void display() {
        System.out.println("Private: " + privateVar);
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
    }
}

public class AccessModifierTest {
```

```
    public static void main(String[] args) {
        AccessExample obj = new AccessExample();
        obj.display();
        // obj.privateVar; // ❌ Not accessible
        System.out.println("Public: " + obj.publicVar);  // ✅ Accessible
        System.out.println("Protected: " + obj.protectedVar);  // ✅ Accessible within package
        System.out.println("Default: " + obj.defaultVar);  // ✅ Accessible within package
    }
}
```

Output:

```
Private: 10
Public: 20
Protected: 30
Default: 40
Public: 20
Protected: 30
Default: 40
```

Key points

- ✅ **Objects** are instances of classes.
- ✅ **Classes** define object properties and behavior.
- ✅ **Inheritance** allows code reuse among classes.
- ✅ **Encapsulation** protects data using access modifiers.
- ✅ **Getter and Setter** methods provide controlled access to private variables.
- ✅ **Access Modifiers** regulate variable/method visibility.

Polymorphism - Method Overloading and Method Overriding

Polymorphism refers to the ability of an object to take multiple forms (message can be displayed in more than one form). It is of two types:

1. **Method Overloading (Compile-time polymorphism)** – Defining multiple methods with the same name but different parameters.
2. **Method Overriding (Runtime polymorphism)** – Redefining a method from a parent class in a child class.

Method Overloading Example

```
class OverloadingExample {
  void show(int a) {
    System.out.println("Integer: " + a);
  }

  void show(double a) {
    System.out.println("Double: " + a);
  }

  void show(String s) {
    System.out.println("String: " + s);
  }

  public static void main(String[] args) {
    OverloadingExample obj = new OverloadingExample();
    obj.show(10);
    obj.show(10.5);
    obj.show("Hello");
  }
```

```
}
```

Method Overriding Example

```
class Parent {
   void display() {
      System.out.println("Display method in Parent");
   }
}

class Child extends Parent {
   @Override
   void display() {
      System.out.println("Display method in Child");
   }

   public static void main(String[] args) {
      Parent obj = new Child();
      obj.display();
   }
}
```

Abstraction Definition and Example

Abstraction is the process of hiding implementation details and showing only functionality. It can be achieved using abstract classes and interfaces.

Example Using Abstract Class

```
abstract class Vehicle {
   abstract void start(); // Abstract method
```

```
    void stop() { // Concrete method
        System.out.println("Vehicle is stopping...");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car is starting with a key...");
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start();
        myCar.stop();
    }
}
```

Interface

An interface is a collection of abstract methods and static constants. It defines a contract that implementing classes must follow.

Example of Interface

```
interface Animal {
    void sound(); // Abstract method
```

```
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();
    }
}
```

Types of Interfaces

1. **Normal Interface** - Contains multiple methods.
2. **Functional Interface** - Contains only one abstract method (used in Lambda expressions).
3. **Marker Interface** - Contains no methods (e.g., Serializable).
4. **Nested Interface** - Interface inside another interface.

Functional Interface Example

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}
```

```
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Calculator obj = (a, b) -> a + b;
        System.out.println("Sum: " + obj.add(5, 10));
    }
}
```

Anonymous Inner Class Explanation and Code

An **Anonymous Inner Class** is a class that does not have a name and is instantiated in a single expression. It is useful when you need a short implementation of an interface or a class.

Example

```
abstract class Person {
    abstract void greet();
}


public class AnonymousClassExample {
    public static void main(String[] args) {
        Person obj = new Person() { // Anonymous Inner Class
            void greet() {
                System.out.println("Hello from Anonymous Class!");
            }
        };
        obj.greet();
    }
}
```

Enum

An **Enum** (Enumeration) is a special class that represents **a group of constants**.

Basic Enum Example

```
enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}


public class EnumExample {
   public static void main(String[] args) {
      Day today = Day.SUNDAY;
      System.out.println("Today is: " + today);
   }
}
```

Enum with Switch Case

```
enum Level {
   LOW, MEDIUM, HIGH;
}


public class EnumSwitchExample {
   public static void main(String[] args) {
      Level level = Level.HIGH;


      switch (level) {
         case LOW:
            System.out.println("Low Level");
            break;
```

```
        case MEDIUM:
            System.out.println("Medium Level");
            break;
        case HIGH:
            System.out.println("High Level");
            break;
    }
  }
}
```

Lambda Expression

Definition: A **Lambda Expression** is a concise way of writing anonymous functions. It is primarily used for implementing functional interfaces.

Uses:

- Reduces boilerplate code.
- Used in functional programming.
- Commonly used in **Streams API, functional interfaces, and event listeners**.

Lambda Expression Example

```
@FunctionalInterface
interface MathOperation {
  int operate(int a, int b);
}


public class LambdaExample {
  public static void main(String[] args) {
    // Lambda expression for addition
    MathOperation addition = (a, b) -> a + b;
```

```
        // Lambda expression for multiplication
        MathOperation multiplication = (a, b) -> a * b;

        System.out.println("Sum: " + addition.operate(5, 10));
        System.out.println("Product: " + multiplication.operate(5, 10));
    }
}
```

Interface (in detail)

Types of Interfaces in Java and Their Code

Java supports different types of interfaces, each with unique characteristics.

1. Normal Interface (Multiple Abstract Methods)

A **normal interface** contains multiple abstract methods that must be implemented by a class.

Example:

```
interface Vehicle {
    void start();
    void stop();
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting...");
    }
```

```
    public void stop() {
        System.out.println("Car is stopping...");
    }
}

public class NormalInterfaceExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start();
        myCar.stop();
    }
}
```

Explanation:

- Vehicle is a normal interface with two abstract methods: start() and stop().
- Car implements Vehicle and provides concrete implementations for both methods.

---

2. Functional Interface (Only One Abstract Method)

A **functional interface** has exactly one abstract method and can be implemented using **lambda expressions**.

Example:

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}

public class FunctionalInterfaceExample {
```

```
    public static void main(String[] args) {
        Calculator obj = (a, b) -> a + b; // Lambda expression
        System.out.println("Sum: " + obj.add(10, 20));
    }
}
```

Explanation:

- @FunctionalInterface ensures the interface contains only one abstract method.
- We use a **lambda expression** to implement the interface concisely.

---

3. Marker Interface (No Methods)

A **marker interface** is an interface that contains **no methods** and is used to indicate a special capability.

Example:

```
interface Marker {}

class Demo implements Marker {
    void show() {
        System.out.println("Class implementing Marker interface");
    }
}

public class MarkerInterfaceExample {
    public static void main(String[] args) {
        Demo obj = new Demo();
        obj.show();
```

```
        if (obj instanceof Marker) {
            System.out.println("This object is of Marker type");
        }
    }
}
```

Explanation:

- Marker interface has no methods.
- Demo implements Marker to indicate that objects of this class have some special behavior.
- instanceof is used to check if obj belongs to Marker.

---

4. Nested Interface (Interface Inside Another Interface)

A **nested interface** is an interface defined inside another interface.

Example:

```
interface OuterInterface {
    void show();

    interface InnerInterface {
        void display();
    }
}

class Demo implements OuterInterface.InnerInterface {
    public void display() {
        System.out.println("Nested Interface Method");
    }
```

```
}


public class NestedInterfaceExample {
   public static void main(String[] args) {
      OuterInterface.InnerInterface obj = new Demo();
      obj.display();
   }
}
```

Explanation:

- OuterInterface contains an **inner interface** InnerInterface.
- Demo implements InnerInterface and provides a method for display().
- NestedInterfaceExample creates an object of Demo using the nested interface reference.

Anonymous Inner Class - Detailed Explanation

An **Anonymous Inner Class** is a class that:

- **Has no name** and is declared and instantiated in a **single expression**.
- **Extends a class or implements an interface** without explicitly defining a separate subclass.
- **Is used for short-term customization** of an existing class.

Example with an Abstract Class

```
abstract class Person {
   abstract void greet();
}


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

```
        Person obj = new Person() { // Anonymous Inner Class
            void greet() {
                System.out.println("Hello from Anonymous Class!");
            }
        };
        obj.greet();
    }
}
```

Explanation:

- Person is an **abstract class** with an abstract method greet().
- Instead of creating a separate subclass, we define an **anonymous inner class** to override greet().
- The class is instantiated immediately using new Person() {...}.

Example with an Interface

```
interface Animal {
    void makeSound();
}

public class AnonymousClassWithInterface {
    public static void main(String[] args) {
        Animal obj = new Animal() { // Anonymous Inner Class implementing an Interface
            public void makeSound() {
                System.out.println("Animal makes a sound!");
            }
        };
```

```
        obj.makeSound();
    }
}
```

Explanation:

- Animal is an **interface** with a single method makeSound().
- Instead of creating a separate class implementing Animal, we use an **anonymous class**.
- The method makeSound() is defined directly within the anonymous class.

Advantages of Anonymous Inner Class

✅ Reduces the need for **separate class declarations**.
✅ Useful for implementing **event listeners and callbacks**.
✅ Helps in quick **one-time use** implementations.

Exception Handling in Java

Exception Handling is a mechanism in Java that handles runtime errors, preventing program crashes and ensuring smooth execution.

Types of Exceptions

There are **two main types** of exceptions in Java:

Exception Type	Description	Examples
Checked Exception	Exceptions that are checked at compile-time. If not handled, the program won't compile.	IOException, SQLException, ClassNotFoundException
Unchecked Exception	Exceptions that occur at runtime. These are **subclasses of RuntimeException**. They occur due to logic errors.	NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

Exception Name	Type (Checked / Unchecked)	Reason
IOException	Checked	Occurs when performing I/O operations like reading a file that does not exist.
SQLException	Checked	Occurs when dealing with databases.
ClassNotFoundException	Checked	Occurs when trying to load a class that is not found.
ArithmeticException	Unchecked	Occurs when dividing by zero.
NullPointerException	Unchecked	Occurs when accessing a method on a null object.
ArrayIndexOutOfBoundsException	Unchecked	Occurs when trying to access an invalid index of an array.

---

try, catch, finally

- **try Block**: Contains the code that may throw an exception.
- **catch Block**: Handles the exception that occurs inside the try block.
- **finally Block**: Always executes, whether an exception occurs or not. It is used for cleanup operations.

Example

```
public class TryCatchFinallyExample {
  public static void main(String[] args) {
    try {
      int num = 10 / 0; // This will cause ArithmeticException
    } catch (ArithmeticException e) {
```

```
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Output

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

Finally block executed

Custom Exception

What is a Custom Exception?

A **custom exception** (also called **user-defined exception**) is created by **extending the Exception class.** It allows programmers to define their own exception types.

Example

```
// Custom Exception Class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}


// Main Class
public class CustomExceptionExample {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
```

```
            throw new InvalidAgeException("Age must be 18 or above.");
        } else {
            System.out.println("Access granted.");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

Output

Caught Exception: Age must be 18 or above.

---

throw vs throws

Difference Between throw and throws

Feature	**throw**	**throws**
Definition	Used to explicitly throw an exception.	Used to declare an exception in a method signature.
Where It Is Used?	Inside a method or block.	In the method signature.
Can It Handle Multiple Exceptions?	No, only one exception at a time.	Yes, multiple exceptions can be declared.

Feature	throw	throws
Example Exception	throw new ArithmeticException("Error message");	void method() throws IOException {}

Example of throw

```
public class ThrowExample {
   static void validate(int age) {
      if (age < 18) {
         throw new ArithmeticException("Not eligible to vote");
      } else {
         System.out.println("Eligible to vote");
      }
   }

   public static void main(String[] args) {
      validate(15);
   }
}
```

Output

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

Example of throws

```
import java.io.*;

class ThrowsExample {
```

```
    static void readFile() throws IOException {
        FileReader file = new FileReader("nonexistent.txt"); // File not found
        BufferedReader br = new BufferedReader(file);
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Exception handled: " + e);
        }
    }
}
```

Output

Exception handled: java.io.FileNotFoundException: nonexistent.txt

---

Feature	**throw**	**throws**
Purpose	Manually throw an exception	Declare exceptions in method signature
Used Inside	Method body	Method signature
Follows With	Exception instance	Exception class
Example	throw new NullPointerException("error");	void method() throws IOException

What is Multithreading?

Multithreading is a **parallel execution** concept in Java that allows multiple threads to run **concurrently** within a single process. It helps in utilizing CPU efficiently.

Advantages of Multithreading

- **Increases efficiency**: Multiple tasks can run in parallel.
- **Better CPU utilization**: No CPU idling, better resource usage.
- **Faster execution**: Threads share memory and resources.
- **Asynchronous processing**: Useful in I/O operations, GUI applications, etc.

What is a Thread in Java?

A **thread** is a **lightweight sub-process** that executes independently. In Java, a thread can be created in two ways:

1. **Extending the Thread class**
2. **Implementing the Runnable interface** (preferred approach)

1. Creating a Thread Using Thread Class

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Thread is running...");
  }

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

2. Creating a Thread Using Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable...");
    }

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

Difference Between Thread and Runnable

Feature	Extending Thread	Implementing Runnable
Inheritance	Cannot extend another class	Can extend other classes
Code Reusability	Less	More
Preferred?	No	Yes

What is a Race Condition in Java?

A **race condition** occurs when multiple threads try to access and modify the same shared resource **simultaneously**, leading to **inconsistent data**.

Example of Race Condition

```
class Counter {
    int count = 0;

    public void increment() {
        count++; // Not thread-safe
```

```
    }
}

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

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

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

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

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.count);
```

```
    }
}
```

How to Prevent Race Condition?

1. **Synchronization (synchronized keyword)**
2. **Using Lock from java.util.concurrent.locks**
3. **Using Atomic variables like AtomicInteger**

Thread Safety in Java

Thread safety ensures that multiple threads can access shared resources **without causing data inconsistency**.

Thread-Safe Code Using Synchronization

```
class SafeCounter {
    private int count = 0;

    public synchronized void increment() { // Synchronized method
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class ThreadSafeExample {
    public static void main(String[] args) {
        SafeCounter counter = new SafeCounter();
```

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

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

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

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Thread Priority and Sleep

Thread Priority

- Each thread has a **priority** ranging from **1 (MIN_PRIORITY)** to **10 (MAX_PRIORITY)**.
- Default priority: 5 (NORM_PRIORITY).

- Higher-priority threads may execute before lower-priority threads.

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread Name: " + Thread.currentThread().getName());
   }

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

      t1.setPriority(Thread.MAX_PRIORITY);
      t2.setPriority(Thread.MIN_PRIORITY);

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

Thread Sleep

- **Pauses execution** for a specified time.
- Thread.sleep(milliseconds)

```
class SleepExample extends Thread {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         try {
            Thread.sleep(1000); // Sleep for 1 second
         } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println(i);
    }
  }

  public static void main(String[] args) {
    SleepExample t1 = new SleepExample();
    t1.start();
  }
}
```

Stream API in Java

The **Stream API** allows processing of collections in a **functional style**.

1. map() Example

```
import java.util.Arrays;
import java.util.List;

public class StreamMapExample {
  public static void main(String[] args) {
    List<String> names = Arrays.asList("Anupam", "Shukla", "Java");
    names.stream().map(String::toUpperCase).forEach(System.out::println);
  }
}
```

2. filter() Example

```
import java.util.Arrays;
```

```
import java.util.List;


public class StreamFilterExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


      numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
   }
}
```

3. reduce() Example

```
import java.util.Arrays;
import java.util.List;


public class StreamReduceExample {
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


      int sum = numbers.stream().reduce(0, Integer::sum);
      System.out.println("Sum: " + sum);
   }
}
```

Collection API

The **Collection API** provides a **framework** for handling and manipulating groups of objects in Java.

Collection Type	Implementation
List	ArrayList, LinkedList
Set	HashSet, TreeSet
Queue	PriorityQueue, Deque
Map	HashMap, TreeMap

StringBuffer vs StringBuilder

Feature	StringBuffer	StringBuilder
Thread-Safe?	Yes	No
Performance	Slower	Faster
Usage	Multi-threaded	Single-threaded

ParallelStream API

```
import java.util.Arrays;

import java.util.List;


public class ParallelStreamExample {

   public static void main(String[] args) {

      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);


      numbers.parallelStream().forEach(System.out::println);

   }

}
```

Stream API: map(), filter(), and reduce() in Java

The **Stream API** in Java is used for processing collections in a **functional and declarative** manner. It allows transformations, filtering, and aggregation of data efficiently.

---

1. map() – Transformation

The map() function is used to **transform** each element of a stream into another form using a **function**. It **does not modify** the original collection but returns a new stream.

Example: Converting Strings to Uppercase

```
import java.util.Arrays;

import java.util.List;


public class StreamMapExample {

   public static void main(String[] args) {

      List<String> names = Arrays.asList("anupam", "shukla", "java");


      // Using map() to convert all names to uppercase

      names.stream()

          .map(String::toUpperCase)

          .forEach(System.out::println);

   }

}
```

Output

ANUPAM

SHUKLA

JAVA

---

2. filter() – Filtering Elements

The filter() function is used to **select elements** that match a given **condition**. It takes a **predicate (boolean function)** and only includes elements that satisfy the condition.

Example: Filtering Even Numbers

```
import java.util.Arrays;

import java.util.List;


public class StreamFilterExample {

   public static void main(String[] args) {

      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);


      // Using filter() to get even numbers

      numbers.stream()

            .filter(n -> n % 2 == 0)

            .forEach(System.out::println);

   }

}
```

Output

```
2

4

6
```

---

3. reduce() – Aggregation/Reduction

The reduce() function is used to **combine all elements** of a stream into a **single result** (e.g., sum, product, max, min, etc.). It takes:

1. **Initial value** (identity)
2. **Binary operator** (function that combines two elements)

Example: Sum of Elements in a List

```
import java.util.Arrays;
import java.util.List;

public class StreamReduceExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Using reduce() to find the sum of all numbers
        int sum = numbers.stream()
                .reduce(0, Integer::sum);

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

Output

```
Sum: 15
```

Function	Purpose	Input	Output
map()	Transforms elements	Function (T → R)	Stream of transformed elements
filter()	Filters elements based on condition	Predicate (boolean function)	Stream of filtered elements
reduce()	Aggregates elements into one value	Binary operator (accumulator)	Single result

---

📌 Java File Handling

What is File Handling?

File handling in Java allows us to perform operations such as **reading, writing, appending, and deleting** files. Java provides the java.io and java.nio packages for file handling.

Types of File Handling in Java

1. **Character Streams** (for text files) → FileReader, FileWriter, BufferedReader, BufferedWriter
2. **Byte Streams** (for binary files) → FileInputStream, FileOutputStream, BufferedInputStream, BufferedOutputStream
3. **NIO (New Input/Output)** (for fast operations) → Files, Paths, Channels

---

Creating a File in Java

The File class is used to create new files.

```
import java.io.File;
import java.io.IOException;


public class CreateFile {
  public static void main(String[] args) {
    try {
      File file = new File("example.txt");
      if (file.createNewFile()) {
        System.out.println("File created: " + file.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
```

```
            e.printStackTrace();
        }
    }
}
```

✅ **Use Case:** When you need to check if a file exists and create it if it doesn't.

Writing to a File

We can write text using FileWriter or BufferedWriter.

🔹 **FileWriter (Character Stream)**

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

public class WriteFile {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, World!");
            writer.close();
            System.out.println("Successfully written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ **Use Case:** When you need **simple writing operations** without buffering.

🔹 **BufferedWriter (More Efficient)**

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

public class BufferedWriteFile {
  public static void main(String[] args) {
    try {
      BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt"));
      writer.write("This is a buffered writer example.");
      writer.newLine(); // Adds a new line
      writer.write("It is faster than FileWriter.");
      writer.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

✅ **Use Case:** When writing **large amounts of data**, as it reduces disk I/O.

Reading from a File

We can read files using FileReader or BufferedReader.

🔹 **FileReader (Character Stream)**

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

```
public class ReadFile {
   public static void main(String[] args) {
      try {
         FileReader reader = new FileReader("example.txt");
         int ch;
         while ((ch = reader.read()) != -1) {
            System.out.print((char) ch);
         }
         reader.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

✅ **Use Case:** When reading small files **character by character**.

🔹 **BufferedReader (More Efficient)**

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

public class BufferedReadFile {
   public static void main(String[] args) {
      try {
         BufferedReader reader = new BufferedReader(new FileReader("example.txt"));
         String line;
```

```
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

✅ **Use Case:** When reading **large files line by line** efficiently.

Appending to a File

To append data, set FileWriter in **append mode** (true).

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

public class AppendFile {
  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter("example.txt", true);
      writer.write("\nAppending this text.");
      writer.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

```
}
```

✅ **Use Case:** When adding new data **without overwriting** existing content.

Deleting a File

The delete() method is used to remove a file.

```
import java.io.File;


public class DeleteFile {
   public static void main(String[] args) {
      File file = new File("example.txt");
      if (file.delete()) {
         System.out.println("Deleted: " + file.getName());
      } else {
         System.out.println("File not found.");
      }
   }
}
```

✅ **Use Case:** When cleaning up unnecessary files.

📌 **Byte Streams (Binary File Handling)**

Writing Binary Data

We use FileOutputStream for binary files.

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


public class WriteBinary {
```

```
    public static void main(String[] args) {
        try {
            FileOutputStream fos = new FileOutputStream("binary.dat");
            fos.write(65); // ASCII for 'A'
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ **Use Case:** Writing images, videos, or binary data.

Reading Binary Data

We use FileInputStream.

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

public class ReadBinary {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("binary.dat");
            int ch;
            while ((ch = fis.read()) != -1) {
                System.out.print((char) ch);
            }
            fis.close();
```

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

✅ **Use Case:** Reading **binary files** efficiently.

📌 **Comparison Table (When to Use What?)**

Feature	FileWriter	BufferedWriter	FileReader	BufferedReader	FileOutputStream	FileInputStream
Text/Binary	Text	Text	Text	Text	Binary	Binary
Performance	Slow	Fast	Slow	Fast	Slow	Fast
Use Case	Writing small text files	Writing large text files	Reading small text files	Reading large text files	Writing binary data	Reading binary data

---

🔹 **Key Takeaways**

✅ Use **BufferedReader/BufferedWriter** for large text files (better performance).
✅ Use **FileInputStream/FileOutputStream** for binary files (images, PDFs, etc.).
✅ Use **FileWriter/FileReader** for small text files.
✅ Always close streams to **prevent memory leaks**.

📌 **InputStream & OutputStream in Java (Detailed Explanation)**

In Java, **InputStream** and **OutputStream** are abstract classes in the java.io package that handle **byte-based** input and output operations. They are used to process **binary data** such as images, videos, PDFs, and audio files.

What is InputStream?

InputStream is an **abstract class** that represents an input stream of **bytes**. It is used to **read data from a source** (file, network, keyboard, etc.).

Common Subclasses of InputStream

Class	Description
FileInputStream	Reads data from a file.
BufferedInputStream	Improves performance by buffering the input.
ByteArrayInputStream	Reads from an array of bytes.
DataInputStream	Reads primitive data types from an input stream.
ObjectInputStream	Reads serialized objects.

Example: Reading from a File using FileInputStream

```
import java.io.FileInputStream;

import java.io.IOException;


public class InputStreamExample {

   public static void main(String[] args) {

      try {

         FileInputStream fis = new FileInputStream("example.txt");

         int data;

         while ((data = fis.read()) != -1) {

            System.out.print((char) data);  // Reads byte by byte

         }

         fis.close();

      } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

🔹 **How it Works?**

- read() reads **one byte at a time**.
- When **end-of-file (EOF)** is reached, it returns -1.
- We cast int to (char) to get readable output.

✅ **Use Case:** When reading binary files such as images, videos, or PDF documents.

What is OutputStream?

OutputStream is an **abstract class** that represents an output stream of **bytes**. It is used to **write data to a destination** (file, network, console, etc.).

Common Subclasses of OutputStream

Class	Description
FileOutputStream	Writes data to a file.
BufferedOutputStream	Improves performance by buffering the output.
ByteArrayOutputStream	Writes to an array of bytes.
DataOutputStream	Writes primitive data types to an output stream.
ObjectOutputStream	Writes serialized objects.

Example: Writing to a File using FileOutputStream

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

public class OutputStreamExample {
```

```
    public static void main(String[] args) {
        try {
            FileOutputStream fos = new FileOutputStream("example.txt");
            String text = "Hello, World!";
            fos.write(text.getBytes());  // Convert String to bytes
            fos.close();
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

🔹 **How it Works?**

- getBytes() converts the **String** into a **byte array**.
- write() writes **each byte** to the file.
- close() releases system resources.

✅ **Use Case:** When writing binary files like images, audio, or encrypted data.

Difference Between InputStream and OutputStream

Feature	**InputStream**	**OutputStream**
Purpose	Reads **bytes** from a source	Writes **bytes** to a destination
Common Methods	read(), available(), close()	write(), flush(), close()
Example Class	FileInputStream	FileOutputStream

Feature	InputStream	OutputStream
Use Case	Reading files, network data, user input	Writing files, sending data over the network

---

Buffered Streams for Performance Improvement

Using BufferedInputStream and BufferedOutputStream can significantly improve performance by **reducing disk I/O operations**.

- **BufferedInputStream Example**

```
import java.io.BufferedInputStream;

import java.io.FileInputStream;

import java.io.IOException;


public class BufferedInputExample {

   public static void main(String[] args) {

      try {

         BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("example.txt"));

         int data;

         while ((data = bis.read()) != -1) {

            System.out.print((char) data);

         }

         bis.close();

      } catch (IOException e) {

         e.printStackTrace();

      }

   }
```

```
}
```

✅ **Use Case:** When reading large files to **reduce disk access time**.

🔹 **BufferedOutputStream Example**

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

public class BufferedOutputExample {
   public static void main(String[] args) {
      try {
         BufferedOutputStream bos = new BufferedOutputStream(new
FileOutputStream("example.txt"));
         String text = "Buffered OutputStream Example";
         bos.write(text.getBytes());
         bos.flush();
         bos.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

✅ **Use Case:** When writing large amounts of data efficiently.

5️⃣ When to Use InputStream vs OutputStream

Scenario	Use
Reading text files	FileReader, BufferedReader
Writing text files	FileWriter, BufferedWriter
Reading binary files	FileInputStream, BufferedInputStream
Writing binary files	FileOutputStream, BufferedOutputStream
Reading network data	DataInputStream, ObjectInputStream
Writing network data	DataOutputStream, ObjectOutputStream

🔹 **Key Takeaways**

✅ **Use InputStream for reading bytes** and **OutputStream for writing bytes**.
✅ **Use BufferedInputStream/BufferedOutputStream for better performance** when handling large files.
✅ **Always close streams** to prevent **memory leaks**.
✅ **Use FileReader and FileWriter for text files**, while **FileInputStream and FileOutputStream are best for binary files**.

📌 **Java Collections Framework (Detailed Explanation)**

What is Java Collections Framework?

The **Java Collections Framework (JCF)** is a unified architecture for **storing and manipulating groups of objects**. It includes **interfaces, classes, and algorithms** to efficiently manage data.

✅ **Why is the Java Collections Framework Needed?**

Before Collections, Java used **arrays** to store multiple elements. However, **arrays have fixed sizes** and lack **efficient operations** (like sorting, searching, or dynamic resizing). Collections solve these problems by providing:

- **Dynamic resizing** (e.g., ArrayList grows automatically)
- **Efficient data retrieval** (e.g., HashMap for key-value pairs)
- **Faster insertion/removal** (e.g., LinkedList uses pointers)

- **Sorting and ordering** (e.g., TreeSet, TreeMap maintain sorted order)

Hierarchy of Java Collections Framework

Collection (Interface)

├── List (Interface) → Allows duplicates, maintains insertion order

│ ├── ArrayList → Fast retrieval, slow deletion

│ ├── LinkedList → Fast insertion/deletion, slow retrieval

│ ├── Vector → Thread-safe version of ArrayList

│ └── Stack → LIFO structure (Last-In-First-Out)

│

├── Set (Interface) → No duplicates allowed

│ ├── HashSet → Unordered, allows `null`

│ ├── LinkedHashSet → Maintains insertion order

│ ├── TreeSet → Sorted order, uses `TreeMap`

│

├── Queue (Interface) → Follows FIFO (First-In-First-Out)

│ ├── PriorityQueue → Elements ordered based on priority

│ ├── Deque (Double-ended queue)

│ ├── ArrayDeque → Fast insertion/deletion, no capacity limit

│

Map (Interface) → Key-Value pairs (not a part of Collection interface)

├── HashMap → Fast access, unordered, allows `null` key

├── LinkedHashMap → Maintains insertion order

├── TreeMap → Sorted keys (Red-Black Tree)

├── HashTable → Synchronized version of HashMap

Java Collections with Code and Important Methods

🔹 **3.1 List Interface (Ordered, Allows Duplicates)**

📌 **ArrayList (Fast Retrieval, Slow Deletion)**

✅ **Use Case:** When you need **fast random access (get operations)** and don't need frequent insertions/deletions.

```
import java.util.*;


public class ArrayListExample {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("Apple");
      list.add("Banana");
      list.add("Cherry");
      list.add("Apple"); // Duplicates allowed

      System.out.println(list); // Output: [Apple, Banana, Cherry, Apple]

      list.remove("Banana");  // Remove by value
      list.remove(1);       // Remove by index

      System.out.println(list); // Output: [Apple, Apple]
   }
}
```

🔹 **Important Methods**

Method	Description
add(E e)	Adds an element to the list
get(int index)	Retrieves an element at a given index
remove(Object o)	Removes the first occurrence of the element
size()	Returns the number of elements in the list
contains(Object o)	Checks if the list contains the element

📌 **LinkedList (Fast Insertion/Deletion, Slow Retrieval)**

✅ **Use Case:** When frequent **insertions and deletions** are required (better than ArrayList for these operations).

```
import java.util.LinkedList;


public class LinkedListExample {
   public static void main(String[] args) {
      LinkedList<Integer> list = new LinkedList<>();
      list.add(10);
      list.add(20);
      list.addFirst(5);  // Insert at head
      list.addLast(30);  // Insert at tail


      System.out.println(list); // Output: [5, 10, 20, 30]


      list.removeFirst(); // Remove first element
      list.removeLast();  // Remove last element
```

```
        System.out.println(list); // Output: [10, 20]
    }
}
```

🔹 **3.2 Set Interface (Unique Elements, No Duplicates)**

📌 **HashSet (Fast Lookup, Unordered)**

✅ **Use Case:** When you need **unique elements** and **don't care about order**.

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("Java");
        set.add("Python");
        set.add("Java"); // Duplicate, will not be added

        System.out.println(set); // Output: [Python, Java] (unordered)
    }
}
```

📌 **TreeSet (Sorted Order, No Duplicates)**

✅ **Use Case:** When you need **sorted unique elements**.

```
import java.util.TreeSet;

public class TreeSetExample {
```

```
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(40);
        set.add(10);
        set.add(30);
        set.add(20);

        System.out.println(set); // Output: [10, 20, 30, 40] (sorted)
    }
}
```

🔹 **3.3 Map Interface (Key-Value Pairs)**

📌 **HashMap (Fast Key Lookup, Unordered)**

✅ **Use Case:** When you need **key-value pairs** with **fast retrieval**.

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Cherry");

        System.out.println(map.get(2)); // Output: Banana
    }
}
```

📌 **TreeMap (Sorted Keys, Red-Black Tree)**

✅ **Use Case:** When you need **sorted key-value pairs**.

```
import java.util.TreeMap;


public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<Integer, String> map = new TreeMap<>();
      map.put(3, "C");
      map.put(1, "A");
      map.put(2, "B");


      System.out.println(map); // Output: {1=A, 2=B, 3=C}
   }
}
```

Summary Table

Collection Type	Ordering	Duplicates Allowed?	Use Case
ArrayList	Yes	Yes	Fast retrieval, dynamic array
LinkedList	Yes	Yes	Fast insertion/deletion
HashSet	No	No	Fast unique element storage
TreeSet	Sorted	No	Unique sorted elements
HashMap	No	Keys: No, Values: Yes	Key-value mapping, fast access
TreeMap	Sorted	Keys: No, Values: Yes	Sorted key-value mapping

---

📌 **Java Collections Framework – Important Methods Table**

Collection Type	Method	Description
List (ArrayList, LinkedList, Vector)	add(E e)	Adds an element to the list
	add(int index, E e)	Inserts an element at a specified index
	get(int index)	Retrieves an element at a specified index
	remove(int index)	Removes the element at a specified index
	remove(Object o)	Removes the first occurrence of a specified element
	set(int index, E e)	Updates an element at a given index
	indexOf(Object o)	Returns the index of the first occurrence of an element
	lastIndexOf(Object o)	Returns the index of the last occurrence of an element
	contains(Object o)	Checks if the list contains the given element
	size()	Returns the number of elements in the list
	isEmpty()	Checks if the list is empty
	clear()	Removes all elements from the list

Collection Type	Method	Description
	sort(Comparator<? super E> c)	Sorts the list based on a comparator
ArrayList Specific	ensureCapacity(int minCapacity)	Increases the capacity of the ArrayList
	trimToSize()	Reduces capacity to current size
LinkedList Specific	addFirst(E e)	Adds an element at the beginning
	addLast(E e)	Adds an element at the end
	removeFirst()	Removes the first element
	removeLast()	Removes the last element
	getFirst()	Retrieves the first element
	getLast()	Retrieves the last element
Set (HashSet, LinkedHashSet, TreeSet)	add(E e)	Adds an element to the set (if not already present)
	remove(Object o)	Removes an element from the set
	contains(Object o)	Checks if an element exists in the set
	size()	Returns the number of elements in the set
	isEmpty()	Checks if the set is empty
	clear()	Removes all elements from the set

Collection Type	Method	Description
TreeSet Specific	first()	Retrieves the first (lowest) element
	last()	Retrieves the last (highest) element
	higher(E e)	Retrieves the next higher element after a given element
	lower(E e)	Retrieves the next lower element before a given element
	ceiling(E e)	Returns the least element ≥ given element
	floor(E e)	Returns the greatest element ≤ given element
	pollFirst()	Retrieves and removes the first element
	pollLast()	Retrieves and removes the last element
Queue (PriorityQueue, ArrayDeque)	offer(E e)	Inserts an element (returns false if full)
	poll()	Retrieves and removes the head element
	peek()	Retrieves the head element without removing it
	remove()	Removes a specific element

Collection Type	Method	Description
	size()	Returns the number of elements in the queue
	clear()	Removes all elements from the queue
Deque (ArrayDeque, LinkedList)	addFirst(E e)	Adds an element at the front
	addLast(E e)	Adds an element at the back
	removeFirst()	Removes the first element
	removeLast()	Removes the last element
	getFirst()	Retrieves the first element
	getLast()	Retrieves the last element
Map (HashMap, LinkedHashMap, TreeMap, Hashtable)	put(K key, V value)	Adds a key-value pair to the map
	get(K key)	Retrieves the value associated with a key
	remove(K key)	Removes a key-value pair
	containsKey(K key)	Checks if a key exists
	containsValue(V value)	Checks if a value exists
	size()	Returns the number of key-value pairs
	isEmpty()	Checks if the map is empty
	clear()	Removes all key-value pairs

Collection Type	Method	Description
HashMap Specific	putIfAbsent(K key, V value)	Adds a key-value pair if the key doesn't exist
	replace(K key, V oldValue, V newValue)	Replaces an existing key's value
	compute(K key, BiFunction<K, V, V> function)	Updates the value of a key based on a function
	merge(K key, V value, BiFunction<V, V, V> function)	Merges two values if a key already exists
TreeMap Specific	firstKey()	Retrieves the first (smallest) key
	lastKey()	Retrieves the last (largest) key
	higherKey(K key)	Retrieves the next higher key
	lowerKey(K key)	Retrieves the next lower key
	ceilingKey(K key)	Returns the least key ≥ given key
	floorKey(K key)	Returns the greatest key ≤ given key
	pollFirstEntry()	Retrieves and removes the first entry
	pollLastEntry()	Retrieves and removes the last entry
Stack (LIFO - Last In, First Out)	push(E e)	Pushes an element onto the stack

Collection Type	Method	Description
	pop()	Removes and returns the top element
	peek()	Returns the top element without removing it
	search(Object o)	Searches for an element and returns its position
	isEmpty()	Checks if the stack is empty

📌 Summary

- **List** → Allows duplicates, maintains insertion order.
- **Set** → No duplicates allowed, does not maintain order (except LinkedHashSet).
- **Queue** → FIFO (First-In-First-Out) operations.
- **Deque** → Double-ended queue, supports insertion/removal from both ends.
- **Map** → Stores key-value pairs.

This table should serve as a **quick reference** while working with Java collections. 🚀
Let me know if you need further explanations or modifications! 😊

Here's a table covering all important **String methods** in Java:

Java String Methods

Method	Description	Example
charAt(int index)	Returns the character at the specified index.	"Hello".charAt(1) // 'e'
length()	Returns the length of the string.	"Hello".length() // 5

Method	Description	Example
toLowerCase()	Converts all characters to lowercase.	"HELLO".toLowerCase() // "hello"
toUpperCase()	Converts all characters to uppercase.	"hello".toUpperCase() // "HELLO"
trim()	Removes leading and trailing spaces.	" hello ".trim() // "hello"
substring(int start, int end)	Extracts a portion of the string.	"Hello".substring(1, 4) // "ell"
replace(char old, char new)	Replaces occurrences of a character.	"hello".replace('l', 'p') // "heppo"
replaceAll(String regex, String replacement)	Replaces all occurrences matching a regex.	"123abc".replaceAll("\\d", "X") // "XXXabc"
equals(String str)	Checks if two strings are equal.	"hello".equals("hello") // true
equalsIgnoreCase(String str)	Checks equality, ignoring case.	"Hello".equalsIgnoreCase("hello") // true
startsWith(String prefix)	Checks if a string starts with a prefix.	"hello".startsWith("he") // true
endsWith(String suffix)	Checks if a string ends with a suffix.	"hello".endsWith("lo") // true
contains(String str)	Checks if a string contains another string.	"hello".contains("ell") // true

Method	Description	Example
indexOf(String str)	Returns the first occurrence index.	"hello".indexOf("l") // 2
lastIndexOf(String str)	Returns the last occurrence index.	"hello".lastIndexOf("l") // 3
isEmpty()	Checks if the string is empty.	"".isEmpty() // true
split(String regex)	Splits a string into an array.	"a,b,c".split(",") // ["a", "b", "c"]
join(String delimiter, String... elements)	Joins multiple strings with a delimiter.	String.join("-", "a", "b", "c") // "a-b-c"
compareTo(String str)	Compares two strings lexicographically.	"apple".compareTo("banana") // -1
compareToIgnoreCase(String str)	Compares lexicographically, ignoring case.	"Apple".compareToIgnoreCase("apple") // 0
matches(String regex)	Checks if a string matches a regex.	"123abc".matches("\\d+") // false
toCharArray()	Converts string to char array.	"hello".toCharArray() // ['h', 'e', 'l', 'l', 'o']
getBytes()	Converts string to byte array.	"hello".getBytes()
format(String format, Object... args)	Formats a string.	String.format("Age: %d", 25) // "Age: 25"
repeat(int count)	Repeats a string multiple times.	"ha".repeat(3) // "hahaha"

Method	Description	Example
strip()	Removes whitespace (better than trim()).	" hello ".strip() // "hello"

Java Arrays Methods

Method	Description	Example
Arrays.toString(arr)	Converts an array to a string.	Arrays.toString(new int[]{1, 2, 3}) // "[1, 2, 3]"
Arrays.asList(arr)	Converts an array to a list.	Arrays.asList("a", "b", "c")
Arrays.sort(arr)	Sorts an array in ascending order.	Arrays.sort(arr)
Arrays.binarySearch(arr, key)	Searches for a key in a sorted array.	Arrays.binarySearch(arr, 3)
Arrays.copyOf(arr, newLength)	Copies an array to a new length.	Arrays.copyOf(arr, 5)
Arrays.copyOfRange(arr, start, end)	Copies a range of an array.	Arrays.copyOfRange(arr, 1, 4)
Arrays.equals(arr1, arr2)	Compares two arrays.	Arrays.equals(arr1, arr2)
Arrays.fill(arr, value)	Fills an array with a value.	Arrays.fill(arr, 7)
Arrays.stream(arr)	Creates a stream from an array.	Arrays.stream(arr).sum()
Arrays.parallelSort(arr)	Sorts an array in parallel (faster).	Arrays.parallelSort(arr)

Method	Description	Example
Arrays.deepToString(arr)	Converts a multidimensional array to a string.	Arrays.deepToString(matrix)
Arrays.deepEquals(arr1, arr2)	Compares two multidimensional arrays.	Arrays.deepEquals(arr1, arr2)
System.arraycopy(src, srcPos, dest, destPos, length)	Copies array elements from one to another.	System.arraycopy(arr1, 0, arr2, 0, arr1.length)