1. Introduction to Java

1.1 History of Java

- Java was developed by **James Gosling** and his team at **Sun Microsystems** in **1991**.
- Initially called **Oak**, later renamed to **Java** in **1995**.
- Designed for small electronic devices but became popular for web and enterprise applications.
- In **2009**, Sun Microsystems was acquired by **Oracle Corporation**, which now owns Java.

1.2 Features of Java

1. **Platform Independent** – Write Once, Run Anywhere (WORA) using JVM.
2. **Object-Oriented** – Everything in Java is based on objects and classes.
3. **Simple** – Easy to learn, syntax similar to C/C++.
4. **Secure** – No explicit pointers, runs inside JVM sandbox, has built-in security features.
5. **Robust** – Strong memory management, exception handling, garbage collection.
6. **Multithreaded** – Supports execution of multiple tasks simultaneously.
7. **High Performance** – Uses Just-In-Time (JIT) compiler.
8. **Distributed** – Supports networking and remote method invocation (RMI).

1.3 Understanding JVM, JRE, and JDK

- **JVM (Java Virtual Machine):**
    - It is a virtual machine that runs Java bytecode.
    - Converts bytecode into machine code (platform-specific).
    - Provides memory management and security.

- **JRE (Java Runtime Environment):**
    - Contains **JVM + core libraries** required to run Java programs.
    - Used for running Java applications (not for developing).

- **JDK (Java Development Kit):**
    - Contains **JRE + development tools** (compiler javac, debugger, etc.).
    - Used for developing Java programs.

1.4 Setting up the Java Environment and IDE

- **Install JDK** from Oracle.
- Set **Environment Variables**:
    - JAVA_HOME → points to JDK folder.
    - Add bin folder to PATH.
- **IDE Options**:
    - **Eclipse** – lightweight, widely used.
    - **IntelliJ IDEA** – powerful, user-friendly, smart code completion.
    - **NetBeans** – official IDE, good for beginners.

1.5 Java Program Structure

Example Program:

```
// Package declaration (optional)
package core;

// Import statement (optional)
import java.util.Scanner;

// Class declaration
public class HelloWorld {
```

```
    // Main method (entry point of program)
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Structure Explanation:

1. **Package** – Organizes classes (like folders).
2. **Import** – Brings other classes/libraries into program.
3. **Class** – Blueprint of objects (must match file name).
4. **Main Method** – public static void main(String[] args) → starting point of execution.
5. **Statements** – Code inside method (e.g., System.out.println).

2. Data Types, Variables, and Operators

2.1 Primitive Data Types in Java

Java has 8 primitive data types, which are the basic building blocks of data handling:

1. byte → 8-bit, range: -128 to 127
2. short → 16-bit, range: -32,768 to 32,767
3. int → 32-bit, commonly used for integers
4. long → 64-bit, large integer values
5. float → 32-bit, single-precision decimal numbers (3.14, 5.6f)
6. double → 64-bit, double-precision decimal numbers (3.14159)
7. char → 16-bit Unicode character (‘a’, ‘A’, ‘$’, ‘1’)
8. boolean → true or false values

2.2 Variable Declaration and Initialization

Variable Declaration: Defining a variable with a data type.

Int age;

Variable Initialization: Assigning a value to the variable.

Int age = 20;

2.3 Operators in Java

Operators are symbols that perform operations on variables and values.

1. Arithmetic Operators

Used for basic mathematical operations.
Operators: + , - , * , / , %

2. Relational Operators

Operators: == , != , > , < , >= , <=

Used to compare two values, result is **true/false**.

3. Logical Operators

Used to combine or reverse conditions (works on booleans).

Operators: && (AND), || (OR), ! (NOT)

4. Assignment Operators

Used to assign values to variables.

Operators: = , += , -= , *= , /= , %=

5. Unary Operators

Works on a **single operand** (variable).

Operators: ++ , -- , + , - , !

6. Bitwise Operators

Work at the **bit-level** (only on integers).
Operators: & , | , ^ , ~ , << , >>

2.4. Type Conversion and Type Casting

Type Conversion (Widening / Implicit Casting)

- Automatically converts smaller type → larger type.
  Example

  int num = 10;

  double d = num; // int automatically converted to double

  System.out.println(d); // 10.0

Type Casting (Narrowing / Explicit Casting)

- Manually converting larger type → smaller type.

  Example

  double d = 9.78;

  int i = (int) d; // explicit casting

  System.out.println(i); // 9

3. Control Flow Statements

3.1 If-Else Statements

- Definition:
  Used for decision-making. Executes a block of code if condition is true, otherwise executes the else block.

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

3.2 Switch-Case Statements

Definition:
Used when you want to select one option from multiple choices. Works better than multiple if-else statements.

```
switch(expression) {
   case value1:
      // code block
      break;
   case value2:
      // code block
      break;
   default:
      // default block
}
```

3.3 Loops in Java

Loops are used to execute a block of code **repeatedly** until a condition is met.

a) For Loop

- Used when the number of iterations is **known**.

```
for (int i = 1; i <= 5; i++) {
   System.out.println("Count: " + i);
}
```

b) While Loop

- Used when the number of iterations is **not known** in advance (condition checked first).

```
int i = 1;
while (i <= 5) {
   System.out.println("Count: " + i);
```

```
    i++;
}
```

c) Do-While Loop

- Similar to while, but condition is checked **after execution** (runs at least once).

```
int i = 1;
do {
    System.out.println("Count: " + i);
    i++;
} while (i <= 5);
```

3.4 Break and Continue Keyword

a) Break

- Exits from a loop immediately, even if condition is true.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    System.out.println(i);
}
// Output: 1 2
```

b) Continue

- Skips the current iteration and moves to the next iteration of the loop.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);
}
// Output: 1 2 4 5
```

4. Classes and Objects

4.1 Defining a Class and Object in Java

Class

- A **class** is a **blueprint** or **template** that defines properties (variables) and behaviors (methods).
- It does not occupy memory until an object is created.

Object

- An **object** is an **instance of a class**.
- Objects represent real-world entities created from a class.

4.2 Constructors and Overloading

Constructor

- A **constructor** is a special method that is **automatically called when an object is created**.
- Constructor name same as class name.
- It has **no return type**.

  Example :-

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

    // Constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }
```

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Aman", 21);
        Student s2 = new Student("Priya", 22);

        s1.display();
        s2.display();
    }
}
```

Constructor Overloading

- Having **multiple constructors** with different parameter lists in the same class.
- Helps in **flexibility** while creating objects.

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

    // Default constructor
    Student() {
        name = "Unknown";
        age = 0;
    }
```

```
    // Parameterized constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();           // calls default constructor
        Student s2 = new Student("Ankit", 23);  // calls parameterized
constructor

        s1.display();
        s2.display();
    }
}
```

4.3 Object Creation and Accessing Members

To create an object:

ClassName obj = new ClassName();

obj.variableName;

```
obj.methodName();
```

4.4 this Keyword

this keyword refers to the **current object** of the class.
Uses:

1. To differentiate between **instance variables** and **local variables** when they have the same name.
2. To call another constructor in the same class.
3. To return the current object.

5. Methods in Java

5.1 Defining Methods

- A **method** in Java is a block of code that performs a specific task.
- It improves **code reusability** and **readability**.
- A method is defined inside a class.

Example:-

```
class MathOperations {
   int add(int a, int b) {
      return a + b;
   }
}
```

5.2 Method Parameters and Return Types

Parameters (arguments) → Values passed into a method.

Return type → Defines what value the method will return.

Void -> no return value.

Any Data type(int, String, double, etc) -> must return that type.

Example:-

```
//method with return type and parameters
Int multiply(int x, int y) {
     Return x * y ;
}
//method with no return type
Public void greet(String name){
    System.out.println(name);
}
```

5.3 Method Overloading

- Method Overloading means **having multiple methods with the same name but different parameter lists** (number or type of parameters).
- It increases **readability** and provides **flexibility**.

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

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

```
public class Main {
    public static void main(String[] args) {
        Display d = new Display();
        d.show(10);       // calls method with int parameter
        d.show("Hello");  // calls method with String parameter
    }
}
```

5.4 Static Methods and Variables.

Static variable: A variable that is shared among all object of the class (common property)

Static method: A method that belongs to the class, not to an object. It can be called without creating an object.

```
class Student {
    String name;
    static String college = "ABC College"; // static variable

    // static method
    static void changeCollege(String newCollege) {
        college = newCollege;
    }

    Student(String n) {
        name = n;
    }
```

```
    void display() {
        System.out.println(name + " studies in " + college);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Rahul");
        Student s2 = new Student("Priya");

        s1.display();
        s2.display();

        // changing static variable via static method
        Student.changeCollege("XYZ University");

        s1.display();
        s2.display();
    }
}
```

6. Object-Oriented Programming (OOPs) Concepts

6.1 Basics of OOP

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects." These objects can contain data (fields/attributes) and methods (functions). The main principles of OOP are:

1. **Encapsulation**

- Wrapping data (variables) and code (methods) into a single unit called a *class*.
- It hides the internal details (data hiding) and exposes only what is necessary through *getters* and *setters*.

2. **Inheritance**
    - Mechanism by which one class (child/subclass) acquires the properties and behaviors of another class (parent/superclass).
    - Promotes code reusability.
3. **Polymorphism**
    - Ability of an object to take multiple forms.
    - Two types:
        - **Compile-time (Method Overloading)**
        - **Run-time (Method Overriding)**
4. **Abstraction**
    - Hiding implementation details and showing only essential features to the user.
    - Achieved using **abstract classes** and **interfaces**.

6.2 Inheritance

Inheritance allows classes to build on existing ones.

1. **Single Inheritance**
    - One class inherits from another.
    - Example: class Dog extends Animal
2. **Multilevel Inheritance**
    - A chain of inheritance.
    - Example: Grandparent → Parent → Child
3. **Hierarchical Inheritance**
    - Multiple classes inherit from the same parent class.

- Example: Dog and Cat both inherit from Animal.

6.3 Method Overriding and Dynamic Method Dispatch

1. **Method Overriding**
    - When a subclass provides its own implementation of a method already defined in the superclass.
    - Achieved at *runtime*.
    - Rules:
        - Method name, return type, and parameters must be the same.
        - Access level cannot be more restrictive than the overridden method.
2. **Dynamic Method Dispatch**
    - Process by which a call to an overridden method is resolved at runtime, not compile time.
    - Achieved using **runtime polymorphism**.

      Example:

      Animal a = new Dog(); // reference of parent, object of child

      a.sound(); // calls Dog's sound() at runtime

7. Constructors and Destructors

7.1 Constructor Types in Java

A **constructor** is a special method in Java that is used to initialize objects. It has the same name as the class and does not have a return type.

- **Default Constructor**:

  A constructor with no parameters.

  If no constructor is defined, the Java compiler automatically provides a default one.

Example:

```
class Car {
        Car() {
      System.out.println("Default Constructor Called");
      }
}
```

Parameterized Constructor:

A constructor that takes arguments to initialize object properties.

Example:

```
class Car {
   String model;
   Car(String m) {
      model = m;
   }
}
```

7.2 Copy Constructor (Emulated in Java)

Java **does not provide a built-in copy constructor** like C++.

But it can be emulated by writing a constructor that takes another object of the same class as a parameter and copies its values.

Example:

```
class Car {
   String model;
   Car(String m) { model = m; }
   // Copy Constructor
   Car(Car c) {
```

```
        this.model = c.model;
    }
}
```

7.3 Constructor Overloading

Having more than one constructor in a class with different parameter lists.

It allows flexibility in object creation.

Example:

```
class Car {
    String model;
    int year;

    Car() { model = "Unknown"; year = 0; }          // Default
    Car(String m) { model = m; year = 0; }          // Parameterized
    Car(String m, int y) { model = m; year = y; }   // Overloaded
}
```

7.4 Object Life Cycle and Garbage Collection

The life cycle of an object in Java includes:

1. **Creation** → An object is created using the new keyword.
2. Car c1 = new Car();
3. **Usage** → The object is used to call methods or access data.
4. **Unreachable State** → When there are no references pointing to an object, it becomes unreachable.
5. **Garbage Collection** → Java automatically reclaims the memory of unreachable objects using **Garbage Collector (GC)**.

    We can request GC by:

    System.gc();

But actual collection is controlled by JVM.

8. Arrays and Strings

8.1 One-Dimensional and Multidimensional Arrays

- **One-Dimensional Array**

  A linear collection of elements of the same type stored in contiguous memory.

  Syntax:

  int[] arr = new int[5]; // 1D Array of size 5

  Example: arr[0], arr[1], arr[2] etc.

Multidimensional Array

- An array of arrays, commonly a **2D array** (matrix).
- Syntax:
- int[][] matrix = new int[3][3]; // 2D Array (3 rows, 3 columns)
- Accessed as: matrix[0][1] etc.

8.2 String Handling in Java

In Java, strings are objects used to represent sequences of characters. There are three main classes:

- **String Class**

  Immutable (cannot be changed once created).

  Example:

  String s = "Hello";

- **StringBuffer Class**

  Mutable (can be modified).

  Thread-safe (synchronized).

  Example:

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // modifies original object

- **StringBuilder Class**

  Mutable like StringBuffer.

  Faster, but **not thread-safe**.

  Example:

  StringBuilder sb = new StringBuilder("Hello");

sb.append(" Java");

8.3 Array of Objects

An array that stores references to objects instead of primitive data types.

Example:

```
class Car {
   String brand;
   Car(String b) { brand = b; }
}

public class Demo {
   public static void main(String[] args) {
      Car[] cars = new Car[3];
      cars[0] = new Car("Toyota");
      cars[1] = new Car("Hyundai");
      cars[2] = new Car("Tata");
      for (Car c : cars) {
         System.out.println(c.brand);
      }
```

```
    }
}
```

8.4 String Methods

Some commonly used **methods of the String class**:

length() → returns the length of the string.

"Hello".length(); // 5

charAt(int index) → returns the character at a given index.

"Hello".charAt(1); // 'e'

substring(int begin, int end) → extracts part of the string.

"HelloWorld".substring(0, 5); // "Hello"

equals(String s) → compares two strings (case-sensitive).

"Java".equals("java"); // false

9. Inheritance and Polymorphism.

9.1 Inheritance: Types and Benefits

Inheritance in Java is a mechanism where one class (child/subclass) acquires the properties and behaviors (fields and methods) of another class (parent/superclass).
It helps in **code reusability** and achieving **polymorphism**.

Types of Inheritance in Java (based on OOP concept):

1. **Single Inheritance** → One class inherits another class.
   Example: Car inherits Vehicle.
2. **Multilevel Inheritance** → A class inherits another class, and then another class inherits it (a chain).
   Example: SportsCar → Car → Vehicle.
3. **Hierarchical Inheritance** → Multiple classes inherit the same parent class.
   Example: Car, Bike, Truck all inherit Vehicle.

4. **Hybrid Inheritance** → Combination of two or more types (not directly supported in Java because of the "diamond problem," but can be achieved using **interfaces**).

Benefits of Inheritance:

- Code reusability (avoid rewriting code).
- Method overriding (achieve runtime polymorphism).
- Logical class hierarchy (real-world modeling).
- Extensibility (easy to add new features).

9.2 Method Overriding

- **Definition:** When a subclass provides its own implementation of a method that is already defined in its superclass.
- **Rules:**
    - Method name, return type, and parameters must be the same.
    - The overridden method cannot have a lower access modifier.
    - @Override annotation is recommended.

Example:

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

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

```
    }
}
```

Here, sound() is overridden in the Dog class.

9.3 Dynamic Binding (Run-Time Polymorphism)

- **Definition:** The process of linking a method call to its method body at **runtime** (not compile-time).
- Happens when **method overriding** is used.
- Achieved through **upcasting** (parent class reference → child class object).

Example:

```
Animal a = new Dog(); // upcasting
a.sound(); // Calls Dog's sound() at runtime
```

Even though the reference is Animal, the actual object (Dog) method is executed at runtime.

9.4 super Keyword and Method Hiding

super keyword:

- Used to refer to the **parent class's methods, variables, or constructor**.
- Helpful when child class overrides a method but still wants to call the parent class method.

Method Hiding:

- If a **static method** is defined in both parent and child classes with the same name, it is **method hiding**, not overriding.
- Which method is called depends on the **reference type** (not object).

Example:

```
class Parent {
   static void show() {
```

```
        System.out.println("Parent show");
    }
}

class Child extends Parent {
    static void show() {
        System.out.println("Child show");
    }
}

Parent p = new Child();
p.show(); // Output: Parent show (method hiding)
```

10.Interfaces and Abstract Classes

10.1 Abstract Classes and Methods

- **Abstract Class** is a class declared with the keyword abstract.
- It **cannot be instantiated** (you cannot create objects directly from it).
- It can contain:
    - **Abstract methods** (methods without a body, only declaration).
    - **Concrete methods** (methods with implementation).
- Abstract classes are used when you want to provide a **base class** with some common functionality, but you also want child classes to **implement specific behaviors**.

10.2 Interfaces: Multiple Inheritance in Java

- **Interface** is like a **contract** that defines methods but does not provide implementation (until Java 8 where default and static methods were introduced).
- Declared using interface keyword.
- A class **implements** an interface.
- **Multiple Inheritance in Java:**
    - Java **does not support multiple inheritance with classes** (to avoid diamond problem).
    - But a class can **implement multiple interfaces** → This is how Java achieves multiple inheritance.

10.3 Implementing Multiple Interfaces

- A class can implement **more than one interface** by separating them with commas.
- The class must **provide implementations** for all methods of all interfaces.

11.Packages and Access Modifiers

11.1 Java Packages

- A **package** in Java is a group of related classes, interfaces, and sub-packages.
- It is used to organize code and avoid name conflicts.
- **Types of Packages:**
    1. **Built-in Packages** → Already available in Java (e.g., java.util, java.io, java.sql).
       Example:
    2. import java.util.Scanner;
    3. **User-defined Packages** → Created by programmers to group their own classes.
       Example:
    4. package mypack;

```
public class Hello {
    public void show() {
        System.out.println("Hello from user-defined package");
    }
}
```

11.2 Access Modifiers

Access modifiers define the **visibility** or **scope** of a class, method, or variable.
There are **4 access modifiers** in Java:

1. **Private**
    - Accessible only within the same class.
    - Not visible to other classes.
2. private int data = 40;
3. **Default (No Modifier)**
    - Accessible only within the same package.
    - Also called *package-private*.
4. int data = 50; // default
5. **Protected**
    - Accessible within the same package and also in **subclasses** (even in different packages through inheritance).
6. protected int data = 60;
7. **Public**
    - Accessible from **anywhere** (all classes, all packages).

public int data = 70;

11.3 Importing Packages

- To use a class from another package, we **import** it using the import keyword.

Ways to import:

1. **Import a single class**
2. import java.util.Scanner;
3. **Import all classes of a package**
4. import java.util.*;
5. **Fully qualified name (without import)**

java.util.Scanner sc = new java.util.Scanner(System.in);

11.4 Classpath in Java

- **Classpath** tells Java **where to look for classes and packages**.
- It is an environment variable or command-line option.

Examples:

- Run program with external class:
- java -cp .;mypackage Hello

(. means current directory, mypackage is the package location)

- If CLASSPATH is not set, Java looks into the **current directory (.)** by default.

12.Exception Handling.

12.1 Types of Exceptions in Java

Exceptions are unwanted events that disrupt the normal flow of a program. They are of **two main types**:

(a) Checked Exceptions (Compile-time exceptions)

- Checked by the **compiler** at compile time.
- Programmer must handle them using **try-catch** or **throws**.

- Examples: IOException, SQLException, ClassNotFoundException.

Example:

```
import java.io.*;


public class CheckedExample {
   public static void main(String[] args) {
      try {
         FileReader fr = new FileReader("file.txt"); // may throw
FileNotFoundException
      } catch (IOException e) {
         System.out.println("File not found: " + e);
      }
   }
}
```

(b) Unchecked Exceptions (Runtime exceptions)

- Not checked at compile time, occur at **runtime**.
- Subclasses of RuntimeException.
- Examples: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException.

Example:

```
public class UncheckedExample {
   public static void main(String[] args) {
      int a = 10 / 0; // ArithmeticException (runtime)
      System.out.println(a);
   }
}
```

12.2 Exception Handling Keywords

1. **try** – Block of code that may throw an exception.
2. **catch** – Used to handle the exception.
3. **finally** – Block that always executes (cleanup code).
4. **throw** – Used to explicitly throw an exception object.
5. **throws** – Declares exceptions a method might throw.

Example:

```
public class ExceptionDemo {
   public static void main(String[] args) {
      try {
         int a = 5 / 0; // risky code
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e);
      } finally {
         System.out.println("Finally block always executes");
      }
   }
}
```

12.3 Custom Exception Classes

We can create our own exception by extending the Exception class (for checked exceptions) or RuntimeException (for unchecked exceptions).

Custom exceptions → User-defined, for specific business logic.

13. Multithreading

13.1 Introduction to Threads

- A **thread** is the smallest unit of a process that can run independently.

- In Java, a thread is a **lightweight sub-process**.
- The **main method** itself runs in the **main thread**.
- Multithreading allows multiple threads to execute concurrently → improves performance.

13.2 Creating Threads in Java

(a) By Extending the Thread Class

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

public class ThreadDemo1 {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      t1.start(); // start() creates a new thread and calls run()
   }
}
```

(b) By Implementing the Runnable Interface

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Thread running: " +
Thread.currentThread().getName());
   }
```

```
}

public class ThreadDemo2 {
   public static void main(String[] args) {
      Thread t1 = new Thread(new MyRunnable());
      t1.start();
   }
}
```

13.3 Thread Life Cycle

A Java thread goes through the following states:

1. **New** → Thread created but not started (new Thread()).
2. **Runnable** → After calling start(), waiting for CPU.
3. **Running** → Thread is executing run() method.
4. **Waiting/Timed Waiting** → Waiting for another thread (sleep(), join(), wait()).
5. **Terminated** → Thread has finished execution.

13.4 Synchronization in Java

- When multiple threads access shared resources, **race conditions** may occur.
- **Synchronization** ensures that only **one thread** accesses a resource at a time.
- Done using the synchronized keyword.

Without synchronization → outputs mix up.
With synchronized → outputs are ordered per thread.

13.5 Inter-thread Communication

- Java provides methods for threads to **communicate**:
    - wait() → makes a thread wait until notified.
    - notify() → wakes up a single waiting thread.
    - notifyAll() → wakes up all waiting threads.
- These methods must be used inside a synchronized block.

14.File Handling

14.1 Introduction to File I/O in Java

- **I/O (Input/Output)** in Java is handled through the **java.io package**.
- File I/O allows programs to **read data from files** and **write data to files**.
- Commonly used classes:
    - File, FileReader, FileWriter
    - BufferedReader, BufferedWriter
    - ObjectOutputStream, ObjectInputStream (for serialization)

14.2 FileReader and FileWriter Classes

- These are **character-oriented** classes.
- FileReader → used to **read data from files (text)**.
- FileWriter → used to **write data to files**.

14.3 BufferedReader and BufferedWriter

- These classes **improve efficiency** by using a **buffer** (faster than direct file access).
- BufferedReader → read text line by line using readLine().
- BufferedWriter → write text efficiently using a buffer.

14.4 Serialization and Deserialization

- **Serialization** → process of saving an object's state into a file.
- **Deserialization** → process of restoring object from file.
- Uses ObjectOutputStream and ObjectInputStream.
- Object class must implement Serializable interface.

15.Collections Framework

15.1 Introduction to Collections Framework

- A **Collection** is an **object that groups multiple elements** into a single unit.
- Java provides the **Collections Framework** (java.util package) to **store, access, and manipulate data efficiently**.
- Benefits:
    - Reduces programming effort.
    - Provides **ready-to-use data structures** (like lists, sets, maps).
    - Provides **algorithms** (sorting, searching, etc.).
- Main interfaces: **List**, **Set**, **Map**, **Queue**.

15.2 List, Set, Map, and Queue Interfaces

Interface	Description	Key Features
List	Ordered collection (allows duplicates)	ArrayList, LinkedList
Set	Collection of unique elements	HashSet, TreeSet
Map	Key-value pairs	HashMap, TreeMap
Queue	First-In-First-Out (FIFO)	PriorityQueue, LinkedList

15.3 ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

1. ArrayList

- Dynamic array, allows duplicates.
- Fast **random access**, slow insert/delete in middle.

```
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Apple"); // duplicates allowed
```

---

2. LinkedList

- Doubly linked list, allows duplicates.
- Fast insert/delete, slower random access.

```
import java.util.LinkedList;

LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(10);
numbers.add(20);
numbers.add(30);
```

---

3. HashSet

- Stores **unique elements only**.
- No guaranteed order.

```
import java.util.HashSet;
```

```
HashSet<String> set = new HashSet<>();
set.add("Red");
set.add("Blue");
set.add("Red"); // duplicate ignored
```

---

4. TreeSet

- Stores **unique elements** in **sorted order**.

```
import java.util.TreeSet;

TreeSet<Integer> set = new TreeSet<>();
set.add(50);
set.add(20);
set.add(30);
System.out.println(set); // Output: [20, 30, 50]
```

---

5. HashMap

- Stores data as **key-value pairs**.
- No guaranteed order.

```
import java.util.HashMap;

HashMap<Integer, String> map = new HashMap<>();
map.put(1, "Luffy");
map.put(2, "Zoro");
map.put(3, "Nami");
System.out.println(map.get(2)); // Output: Zoro
```

6. TreeMap

- Stores **key-value pairs** in **sorted order of keys**.

```
import java.util.TreeMap;

TreeMap<Integer, String> map = new TreeMap<>();
map.put(3, "Sanji");
map.put(1, "Luffy");
map.put(2, "Zoro");
System.out.println(map);
```

15.4 Iterators and ListIterators

- **Iterator** → used to traverse **any Collection** (List, Set).
- Methods:
    - hasNext() → checks if more elements exist
    - next() → returns next element
    - remove() → removes the last returned element

16.**Java Input/Output (I/O)**

16.1 Streams in Java

- A **stream** is a **sequence of data** that can be read from or written to.
- Java provides **InputStream** and **OutputStream** classes for handling **byte-oriented I/O**.
- Streams are used to **read from sources** (files, network, memory) and **write to destinations**.

16.2 Reading and Writing Data Using Streams

- In Java, **streams** are used to read and write **data sequentially**.

- There are **two types of streams**:
    1. **Byte Streams** – Handle data **byte by byte**. Classes:
        - InputStream → read bytes from a source.
        - OutputStream → write bytes to a destination.
        - Common subclasses: FileInputStream, FileOutputStream.
    2. **Character Streams** – Handle **characters**. Classes:
        - Reader → read characters.
        - Writer → write characters.
        - Common subclasses: FileReader, FileWriter.

16.3 Handling File I/O Operations

- Java provides the **java.io.File class** to **perform operations on files and directories**.

Common File I/O Operations:

1. **Creating a file** – File.createNewFile() creates a new file if it doesn't exist.
2. **Checking file properties** – Methods like canRead(), canWrite(), exists(), length() help check permissions, existence, and size.
3. **Deleting a file** – File.delete() removes a file from the system.
4. **Listing directory contents** – list() and listFiles() return file names or file objects in a directory.