History of Java

- Developed by James Gosling and his team at Sun Microsystems (1991).
- Initially called Oak (named after a tree outside Gosling's office).
- Later renamed Java (in 1995), inspired by Java coffee.
- In 2009, Sun Microsystems was acquired by Oracle.
- Motto: "Write Once, Run Anywhere (WORA)".
- Java quickly became popular due to its portability, security, and reliability.

Features of Java

1. Platform Independent – Java bytecode runs on JVM.
2. Object-Oriented – Supports OOPs concepts.
3. Simple & Familiar – Similar to C/C++.
4. Secure – No pointers.
5. Robust – Strong memory management, exception handling.
6. Multithreaded – Can run multiple tasks simultaneously.
7. Portable – Bytecode is machine-independent.
8. High Performance – Uses JIT compiler.

JVM, JRE, and JDK

- **JVM** (Java Virtual Machine): Executes Java bytecode, provides platform independence.
- **JRE** (Java Runtime Environment): JVM + Libraries; used to run programs.
- **JDK** (Java Development Kit): JRE + Compiler + Tools; used to develop programs.

Formula:

JDK = JRE + Development Tools

JRE = JVM + Libraries

Setting up Java Environment & IDE

1. Install JDK from Oracle or OpenJDK.

2. Set JAVA_HOME and PATH environment variables.

3. Verify installation using: java -version, javac -version.

4. Install IDE (Eclipse, IntelliJ IDEA, or VS Code).

Java Program Structure

Example Java Program:

```
package core_practice;
import java.util.Scanner;

public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello, World!");
}
public void greet(String name) {
System.out.println("Hello, " + name);
}
}
```

Explanation:

- Package: Groups related classes.

- Class: Blueprint of an object.

- Method: Block of code that performs a task.

- main(): Entry point of application.

- Statements: Instructions inside methods

Primitive Data Types in Java

Java has 8 primitive data types:

1. byte: 8-bit integer.

2. short: 16-bit integer.

3. int: 32-bit integer, commonly used.

4. long: 64-bit integer, used when int is not enough

5. float: 32-bit decimal, single precision

6. double: 64-bit decimal, double precision (default for decimals)

7. char: 16-bit Unicode character (e.g., 'A', '1')

8. boolean: Represents true/false values

Variable Declaration and Initialization

- Declaration: int age;

- Initialization: age = 25;

- Combined: int age = 25;

Operators in Java

1. Arithmetic Operators

- `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus)

Example: int sum = a + b;

2. Relational Operators

- `==` (equal), `!=` (not equal), `>` (greater), `<` (less), `>=`, `<=`

Example: if (a > b)

3. Logical Operators

- `&&` (AND), `||` (OR), `!` (NOT)

Example: if (a > 10 && b < 5)

4. Assignment Operators

- `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Example: x += 5; // same as x = x + 5

5. Unary Operators

- `++` (increment), `--` (decrement), `+` (unary plus), `-` (unary minus), `!` (logical NOT)

Example: count++

Type Conversion and Casting

-**Type Conversion (Widening)**

-Automatic conversion from smaller to larger data type

Example:

int num = 10;

double d = num; // int to double

-Type Casting (Narrowing)

Explicit conversion from larger to smaller type

Example:

double d = 10.5;

int num = (int) d; // double to int

If-Else Statements

If-Else statements are used to execute a block of code based on a condition.

Example:

int num = 10;

if (num > 0) {

System.out.println("Positive number");

} else {

System.out.println("Non-positive number");

```
}
```

Switch Case Statements

Switch statement is used to execute one block of code from multiple options.

Example:

```
int day = 3;

switch(day) {

case 1:
System.out.println("Monday");
break;

case 2:
System.out.println("Tuesday");
break;

case 3:
System.out.println("Wednesday");
break;

default:
System.out.println("Invalid day");

}
```

Loops (For, While, Do-While)

Loops are used to execute a block of code repeatedly.

For Loop:

```
for (int i = 1; i <= 5; i++) {

System.out.println(i);

}
```

While Loop:

```
int i = 1;

while (i <= 5) {

System.out.println(i);
```

```
i++;
}
```

Do-While Loop:

```
int j = 1;
do {
System.out.println(j);
j++;
} while (j <= 5);
```

Break and Continue Keywords

Break is used to exit a loop or switch statement prematurely.

Continue is used to skip the current iteration of a loop and continue with the next iteration.

Example with Break:

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

Example with Continue:

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

Defining a Class and Object in Java

Class:

A class in Java is a blueprint or template from which objects are created. It defines the properties (fields) and methods of objects.

Syntax:

```
class Car {
   // Fields (properties)
   String color;
   String model;

   // Method
   void displayDetails() {
      System.out.println("Model: " + model + ", Color: " + color);
   }
}
```

Object:

An object is an instance of a class. It is created from the class blueprint and can access the class fields and methods.

Example:

```
public class Main {
   public static void main(String[] args) {
      Car car1 = new Car(); // Creating object
      car1.model = "Honda";
      car1.color = "Red";
```

```
        car1.displayDetails(); // Accessing method
    }
}
```

Constructors and Overloading

Constructor:

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

Syntax:

```
class Car {
    String color;
    String model;

    // Constructor
    Car(String model, String color) {
        this.model = model;
        this.color = color;
    }
}
```

Constructor Overloading:

You can have multiple constructors with different parameters in a class. This is called constructor overloading.

Example:

```
class Car {
    String color;
```

```
    String model;

    Car() { // Default constructor
        model = "Unknown";
        color = "Black";
    }

    Car(String model) { // Parameterized constructor
        this.model = model;
        this.color = "Black";
    }

    Car(String model, String color) { // Parameterized constructor
        this.model = model;
        this.color = color;
    }
}
```

Object Creation and Accessing Members

Object Creation:

Objects can be created using the `new` keyword.

Syntax:

```
Car car1 = new Car();  // Default constructor
Car car2 = new Car("Honda"); // Parameterized constructor
Car car3 = new Car("Toyota", "Red"); // Parameterized constructor
```

Accessing Members:

You can access fields and methods of an object using the dot (`.`) operator.

Example:

System.out.println(car3.model); // Accessing field

car3.displayDetails(); // Calling method

The `this` Keyword

* Refers to the current object of a class.

* Used to avoid naming conflicts between class fields and parameters.

* Can be used to call another constructor in the same class.

Example:

```
class Car {
   String model;
   String color;

   Car(String model, String color) {
      this.model = model; // Refers to current object's field
      this.color = color;
   }

   void print() {
      System.out.println("Model: " + this.model + ", Color: " + this.color);
   }
}
```

Defining Methods

Method:

A method in Java is a block of code that performs a specific task. Methods allow code reuse and modular programming.

Syntax:

```
returnType methodName(parameters) {
   // method body
}
```

Example:

```
class Calculator {
   int addNumbers(int a, int b) {
      return a + b;
   }
}
public class Main {
   public static void main(String[] args) {
      Calculator calc = new Calculator();
      int sum = calc.addNumbers(5, 10);
      System.out.println("Sum: " + sum);
   }
}
```

Method Parameters and Return Types

Parameters:

* Variables passed to a method to provide input.

* Methods can have zero or more parameters.

Return Type:

* Specifies the type of value a method returns.

* If a method does not return any value, use `void`.

Example:

```
class Calculator {
   // Method with parameters and return type
   int multiply(int a, int b) {
      return a * b;
   }

   // Method without return type
   void displayMessage() {
      System.out.println("Hello from Calculator!");
   }
}
```

Method Overloading

Method Overloading:

* Defining multiple methods in the same class with the same name but different parameter lists.

* Return type can be the same or different.

Example:

```
class Calculator {
```

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

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
Calculator calc = new Calculator();
System.out.println(calc.add(5, 10));       // Calls first method
System.out.println(calc.add(5.5, 4.5));    // Calls second method
System.out.println(calc.add(1, 2, 3));     // Calls third method
```

Static Methods and Variables

Static Variable:

* Belongs to the class rather than an instance.
* Shared by all objects of the class.

Static Method:

* Can be called without creating an object.

* Can access only static members directly.

Example:

```
class Calculator {
   static int count = 0; // Static variable

   static int square(int num) { // Static method
      count++;
      return num * num;
   }
}

public class Main {
   public static void main(String[] args) {
      System.out.println(Calculator.square(5));
      System.out.println("Method called " + Calculator.count + " times");
   }
}
```

Basics of OOP

Java is an Object-Oriented Programming (OOP) language. The four main principles of OOP are:

Encapsulation:

* Wrapping data (fields) and methods into a single unit (class).
* Use of **private** fields and **public getters/setters** to protect data.

Example:

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

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Inheritance:

Mechanism where one class acquires the properties (fields) and behaviors (methods) of another class.

Example:

```
class Vehicle {
   void display() {
      System.out.println("This is a vehicle");
   }
}

class Car extends Vehicle {
   void displayCar() {
      System.out.println("This is a car");
```

```
    }
}
```

Polymorphism:

* Ability of a method or object to take multiple forms.

* Two types:
  Compile-time (method overloading)
  Runtime (method overriding)

Abstraction:

* Hiding implementation details and showing only functionality.

* Achieved using abstract classes and interfaces.

Example:

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Inheritance Types

Single Inheritance

* A class inherits from one superclass.

```
class Animal {}
class Dog extends Animal {}
```

Multilevel Inheritance

* A class inherits from a subclass, forming a chain.

```
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {}
```

Hierarchical Inheritance

* Multiple subclasses inherit from a single superclass.

```
class Vehicle {}
class Car extends Vehicle {}
class Bike extends Vehicle {}
```

Method Overriding and Dynamic Method Dispatch

Method Overriding

* A subclass provides a specific implementation of a method already defined in its superclass.

* The method signature must be the same.

Example:

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

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

Dynamic Method Dispatch

* Runtime polymorphism achieved by overriding methods.

* Reference variable of superclass can point to subclass object.

Example:

```
Animal myAnimal = new Dog();
myAnimal.sound(); // Calls Dog's overridden method
```

Constructor Types

Default Constructor

* A constructor with no parameters.
* Java provides a default constructor if none is defined.

Example:

```
class Car {
   String model;

   Car() { // Default constructor
      model = "Unknown";
   }

   void display() {
      System.out.println("Model: " + model);
   }
}
```

Parameterized Constructor

* Constructor that takes arguments to initialize an object with specific values.

Example:

```
class Car {
   String model;

   Car(String model) { // Parameterized constructor
```

```
        this.model = model;
    }

    void display() {
        System.out.println("Model: " + model);
    }
}
```

Copy Constructor

* Java does not provide a built-in copy constructor, but it can be emulated by creating a constructor that takes an object of the same class.

Example:

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    // Copy constructor
    Car(Car c) {
        this.model = c.model;
    }

    void display() {
        System.out.println("Model: " + model);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota");
        Car car2 = new Car(car1); // Using copy constructor
        car2.display();
    }
}
```

Constructor Overloading

* Multiple constructors in the same class with different parameter lists.

Example:

```
class Car {
    String model;
    String color;

    Car() { // Default
        model = "Unknown";
        color = "Black";
    }

    Car(String model) { // Single parameter
        this.model = model;
        color = "Black";
    }
```

```
    Car(String model, String color) { // Two parameters
        this.model = model;
        this.color = color;
    }
}
```

Object Life Cycle

1. Creation: Using `new` keyword.
2. Usage: Accessing methods and fields.
3. Eligibility for GC: When object has no references.
4. Destruction: Memory reclaimed by Garbage Collector.

Garbage Collection

* Automatic memory management in Java.
* The `finalize()` method can be overridden for cleanup before GC.

Example:

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    protected void finalize() {
        System.out.println("Car object is garbage collected");
```

```
    }

    public static void main(String[] args) {
        Car car1 = new Car("Honda");
        Car car2 = new Car("Toyota");
        car1 = null; // eligible for GC
        car2 = null; // eligible for GC
        System.gc(); // Request JVM to run garbage collector
    }
}
```

One-Dimensional and Multidimensional Arrays

One-Dimensional Array

* Stores elements of the same type in a single row.

Example:

```
int[] numbers = new int[5]; // Declaration
numbers[0] = 10;
numbers[1] = 20;

for(int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Multidimensional Array

* Array of arrays, often used as matrix.

Example:

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};


for(int i = 0; i < matrix.length; i++) {
   for(int j = 0; j < matrix[i].length; j++) {
      System.out.print(matrix[i][j] + " ");
   }
   System.out.println();
}
```

String Handling in Java

String Class

* Immutable objects.
* Supports many methods like `length()`, `charAt()`, `substring()`, `concat()`, `equals()`, etc.

Example:

```
String str = "Hello";
System.out.println(str.length());       // 5
System.out.println(str.charAt(1));     // e
System.out.println(str.substring(1,4));// ell
```

StringBuffer Class

* Mutable sequences of characters.
* Supports methods like `append()`, `insert()`, `delete()`, `reverse()`, etc.

Example:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

StringBuilder Class:

* Similar to StringBuffer but not synchronized (faster in single-threaded programs).

Example:

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" Programming");
System.out.println(sb); // Java Programming
```

Array of Objects

Example:

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

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

   // Method to display student details
```

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

public class Main {
    public static void main(String[] args) {
        // Create an array of Student objects
        Student[] students = new Student[2];
        students[0] = new Student("Alice", 20);
        students[1] = new Student("Bob", 22);

        // Using a normal for loop to iterate over the array
        for (int i = 0; i < students.length; i++) {
            students[i].display();
        }
    }
}
```

String Methods

Method	Description
length()	Returns the length of the string
charAt(int)	Returns character at specified index
substring(a,b)	Returns substring from index a to b-1
concat(String)	Concatenates two strings
equals(String)	Compares two strings for equality
compareTo(String)	Compares two strings lexicographically

toUpperCase()	Converts string to uppercase
toLowerCase()	Converts string to lowercase

Inheritance Types and Benefits

Types of Inheritance

Single Inheritance: A class inherits from one superclass.

```
class Animal {}
class Dog extends Animal {}
```

Multilevel Inheritance: A class inherits from a subclass, forming a chain.

```
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {}
```

Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.

```
class Vehicle {}
class Car extends Vehicle {}
class Bike extends Vehicle {}
```

Benefits of Inheritance

- **Code Reusability**: Reuse fields and methods of existing classes.
- **Method Overriding**: Subclass can provide specific implementation.
- **Extensibility**: Easy to add new features without modifying existing code.
- **Polymorphism**: Supports dynamic method dispatch.

Method Overriding

- Occurs when a subclass provides a specific implementation for a method already defined in its superclass.
- Method signature must be the same.
- Supports **runtime polymorphism**.

Example:

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

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

public class Main {
   public static void main(String[] args) {
      Animal a = new Dog();
      a.sound(); // Calls Dog's overridden method
   }
}
```

Dynamic Binding (Run-Time Polymorphism)

- The JVM decides at **runtime** which method to invoke, based on the object type.
- Achieved through **method overriding** and **superclass reference to subclass object**.

Example:

```
Animal a1 = new Animal();
Animal a2 = new Dog();

a1.sound(); // Animal's method
a2.sound(); // Dog's method (dynamic binding)
```

Super Keyword and Method Hiding

Super Keyword

- Refers to the immediate superclass.
- Used to access superclass methods, constructors, or variables.

Example:

```
class Animal {
   void display() {
      System.out.println("Animal class method");
   }
}

class Dog extends Animal {
   void display() {
      super.display(); // Calls superclass method
      System.out.println("Dog class method");
```

```
    }
}
```

Method Hiding

- Occurs when a **static method** in a subclass has the same name as a static method in the superclass.
- Unlike overriding, **method hiding** is resolved at compile time.

Example:

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

class Dog extends Animal {
    static void show() {
        System.out.println("Dog static method");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.show(); // Calls Animal's static method (method hiding)
        Dog.show(); // Calls Dog's static method
    }
}
```

Abstract Classes and Methods

- **Abstract Class**: A class that cannot be instantiated and may contain abstract methods (without implementation).
- **Abstract Method**: A method declared without a body; must be implemented by subclasses.

Syntax:

```
abstract class Shape {
    abstract void draw(); // Abstract method

    void display() {
        System.out.println("This is a shape");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
        s.display();
    }
```

```
}
```

Key Points:

- Abstract classes can have both abstract and concrete methods.
- Cannot create objects of abstract classes.
- Subclasses must implement all abstract methods.

Interfaces: Multiple Inheritance in Java

- **Interface**: A reference type in Java, similar to a class, but can contain only abstract methods (before Java 8) and constants.
- Java supports multiple inheritance through **interfaces**, not classes.

Syntax:

```
interface Drawable {
    void draw();
}


interface Printable {
    void print();
}
```

Implementing Multiple Interfaces

```
class Graphic implements Drawable, Printable {
    public void draw() {
        System.out.println("Drawing graphic");
    }
```

```
    public void print() {
System.out.println("Printing graphic");
}
}
```

```
public class Main {
public static void main(String[] args) {
Graphic g = new Graphic();
g.draw();
g.print();
}
}
```

Key Points:

-A class can implement multiple interfaces, enabling multiple inheritance.

-All interface methods must be implemented in the class.

-Interfaces cannot have constructors.

Java Packages: Built-in and User-Defined Packages

- A package in Java is a collection of classes, interfaces, and sub-packages that are grouped together.

- **Built-in Packages**: These are provided by Java itself. Example: java.util, java.io, java.sql, etc.

- **User-defined Packages**: Developers can create their own packages to organize related classes.

Example:

```
package myPackage;
public class MyClass { ... }
```

Access Modifiers: Private, Default, Protected, Public

- **Private**: The member is accessible only within the same class.

- **Default** (no modifier): The member is accessible only within the same package.

- **Protected**: The member is accessible within the same package and by subclasses (even if they are in different packages).

- **Public**: The member is accessible from any other class, anywhere.

.

Importing Packages and Classpath.

- Importing Packages:

To use a class from another package, we use the import statement.

Example: import java.util.Scanner;

- **Classpath**:

The classpath tells the JVM where to find user-defined classes and packages.

It can be set using the -cp option in the command line or by setting the CLASSPATH environment variable.

Types of Exceptions: Checked and Unchecked

- **Checked Exceptions**: Checked at compile-time. Must be handled using try-catch or declared using throws.

Examples: IOException, SQLException, ClassNotFoundException.

- **Unchecked Exceptions**: Occur at runtime. Usually caused by programming errors.

Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

Exception Handling Keywords

- **try**: Defines a block of code where exceptions may occur.

- **catch**: Block that handles exceptions.

- **finally**: Always executes whether exception occurs or not (used for resource cleanup).

- **throw**: Used to explicitly throw an exception object.

Example: throw new IOException("File not found");

- **throws**: Declares exceptions that a method may throw.

Example:

public void readFile() throws IOException { ... }

Custom Exception Classes

Java allows developers to create their own exception classes by extending Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).

Example:

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

public class TestCustomException {

public static void main(String[] args) {

try {

int age = 15;

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or above to vote");

}

System.out.println("You can vote!");

} catch (InvalidAgeException e) {

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

}

}

}

Output:
Caught Exception: Age must be 18 or above to vote

Introduction to Threads

- A thread in Java is a lightweight process that runs concurrently with other threads.

- Threads are used to perform multiple tasks simultaneously (multithreading).

- Example: Downloading a file while playing music in the same program.

Creating Threads

There are two main ways to create threads in Java:

1. **Extending Thread class:**

```
class MyThread extends Thread {
public void run() {
System.out.println("Thread is running...");
}
}
MyThread t1 = new MyThread();
t1.start();
```

2. **Implementing Runnable interface:**

```
class MyRunnable implements Runnable {
public void run() {
System.out.println("Thread is running...");
}
}
Thread t1 = new Thread(new MyRunnable());
t1.start();
```

Thread Life Cycle

A thread goes through the following states:

- New: Thread is created but not started.

- Runnable: After calling start(), waiting for CPU scheduling.

- Running: When the thread is executed by the CPU.

- Waiting/Blocked: Thread is inactive temporarily.

- Terminated: Thread has finished execution.

Synchronization

- Synchronization in Java ensures that only one thread can access a shared resource at a time.

- It prevents data inconsistency when multiple threads operate on shared objects.

Inter-thread Communication

- Mechanism by which threads communicate with each other.

- Java provides methods like wait(), notify(), and notifyAll() in Object class.

Example:

- wait(): Causes the current thread to wait until another thread invokes notify().

- notify(): Wakes up a single waiting thread.

- notifyAll(): Wakes up all threads waiting on the object

Introduction to File I/O in Java (java.io package)

Java provides the java.io package for input and output operations (I/O).

File I/O allows reading data from files and writing data to files.

Common classes: FileReader, FileWriter, BufferedReader, BufferedWriter, ObjectInputStream, ObjectOutputStream.

FileReader and FileWriter Classes

FileReader → Reads character data from files.

FileWriter → Writes character data to files.

Example (FileReader):

```
import java.io.FileReader;

public class FileReadExample {
   public static void main(String[] args) throws Exception {
      FileReader fr = new FileReader("file.txt");
      int i;
      while((i = fr.read()) != -1) {
         System.out.print((char)i);
      }
   }
}
```

Example (FileWriter):

```
import java.io.FileWriter;
public class FileWriteExample {
   public static void main(String[] args) throws Exception {
      FileWriter fw = new FileWriter("file.txt");
      fw.write("Hello Java FileWriter!");
   }
}
```

BufferedReader and BufferedWriter

These classes improve efficiency by using buffers.

BufferedReader → Reads text line by line.
BufferedWriter → Writes text efficiently.

Example (BufferedReader):

```
import java.io.*;

public class BufferedReaderExample {
   public static void main(String[] args) throws Exception {
      FileReader fr = new FileReader("file.txt");
      BufferedReader br = new BufferedReader(fr);
      String line;
      while((line = br.readLine()) != null) {
         System.out.println(line);
      }
      br.close();
   }
}
```

Example (BufferedWriter):

```
import java.io.*;
public class BufferedWriterExample {
   public static void main(String[] args) throws Exception {
      FileWriter fw = new FileWriter("file.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      bw.write("Buffered Writer Example");
      bw.newLine();
      bw.write("Another line");
      bw.close();
```

```
    }
}
```

Serialization and Deserialization

Serialization → Converts an object into a byte stream (for saving or sending).

Deserialization → Converts the byte stream back into an object.

Class must implement the Serializable interface.

Example:

```
import java.io.*;

// Class must be Serializable
class Student implements Serializable {
    int id;
    String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class SerializationExample {
    public static void main(String[] args) throws Exception {
        Student s1 = new Student(101, "John");

        // Serialization
```

```
    FileOutputStream fos   = new FileOutputStream("student.ser")
   ObjectOutputStream oos = new ObjectOutputStream(fos);


      oos.writeObject(s1);

      oos.close();


      // Deserialization
      FileInputStream fos = new FileInputStream("student.ser")

      ObjectInputStream ois = new ObjectInputStream(fos);

      Student s2 = (Student) ois.readObject();

      ois.close();


      System.out.println("Deserialized Student: " + s2.id + " " + s2.name);

   }
}
```

Introduction to Collections Framework

- The Collections Framework in Java provides a set of classes and interfaces for storing and manipulating groups of objects.
- It contains interfaces like List, Set, Map, and Queue, and their concrete implementations.

Core Interfaces

- **List**: Ordered collection that allows duplicate elements. Examples: ArrayList, LinkedList.
- **Set**: Collection that does not allow duplicate elements. Examples: HashSet, TreeSet.
- **Map**: Key-value pairs; keys are unique, values may be duplicate. Examples: HashMap, TreeMap.
- **Queue**: Follows FIFO (First In First Out) order. Examples: PriorityQueue, LinkedList.

Common Implementations

- **ArrayList**: Resizable array, fast random access, slower insertions/deletions.

- **LinkedList**: Doubly linked list, faster insertions/deletions, slower random access.

- **HashSet**: Stores unique elements, no guaranteed order.

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

- **HashMap**: Stores key-value pairs, keys are unique, no ordering.

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

Iterators and ListIterators

- **Iterator**: Used to traverse elements of a collection sequentially.

Methods:

- **hasNext**(): returns true if more elements exist.

- **next**(): returns the next element.

- **remove**(): removes the last returned element.

- **ListIterator**: A bidirectional iterator for lists (ArrayList, LinkedList).

Additional methods:

- **hasPrevious**(): checks if there is a previous element.

- **previous**(): returns the previous element.

Streams in Java

- A stream is a sequence of data that can be read from a source or written to a destination.

- Java provides two types of streams:

1. **InputStream** → Used to read data (input).

2. **OutputStream** → Used to write data (output).

- Streams are part of the java.io package.

InputStream and OutputStream

- **InputStream**: Abstract class for reading byte data.

Common subclasses: FileInputStream, ObjectInputStream.

- **OutputStream**: Abstract class for writing byte data.

Common subclasses: FileOutputStream,ObjectOutputStream.

Reading Data Using Streams

Example using FileInputStream:

```
FileInputStream fis = new FileInputStream("data.txt");
int i;
while((i = fis.read()) != -1) {
System.out.print((char)i);
}
fis.close();
```

Writing Data Using Streams

Example using FileOutputStream:

```
FileOutputStream fos = new FileOutputStream("data.txt");
String str = "Hello Java Streams!";
byte[] b = str.getBytes();
fos.write(b);
fos.close();
```

Handling File I/O Operations

- Always close streams after use to free resources.

- Use try-with-resources for automatic closing:

```
try (FileInputStream fis = new FileInputStream("data.txt")) {
int i;
while((i = fis.read()) != -1) {
System.out.print((char)i);
}
} catch(Exception e) {
e.printStackTrace();
}
```

- Buffered streams (BufferedInputStream, BufferedOutputStream) improve performance by reducing I/O calls.