Module – 6

Core Java

Name : Abhay Gajjar

1.) Introduction to Java :

- **Theory :**

1) History of Java :

- Java is a high-level, class-based, object-oriented programming language that was designed to have as few implementation dependencies as possible.
- It was developed by James Gosling and his team at Sun Microsystems in the mid-1990s. The project was initially called "Oak" after an oak tree that stood outside Gosling's office.
- It was later renamed "Java" after a type of coffee from Indonesia, which reflected the language's ability to provide a refreshing approach to programming.

- Java was officially released in 1995, and since then, it has undergone significant evolution. The language's promise of "Write Once, Run Anywhere" (WORA) made it particularly appealing for web applications and enterprise solutions.
- Oracle Corporation acquired Sun Microsystems in 2010, and since then, Oracle has maintained Java, ensuring its continued growth and relevance in the software industry.

2) Features of Java :

- **Platform Independent:** One of the most significant features of Java is its platform independence. Java programs are compiled into bytecode that can run on any device equipped with the Java Virtual Machine (JVM). This allows developers to write code once and run it anywhere without modification.
- **Object-Oriented**: Java follows the object-oriented programming paradigm, which means it focuses on objects and classes. This approach makes it easier to manage and organize code, leading to better software design and maintainability.

- **Simple**: Java was designed to be easy to learn and use. Its syntax is clean and straightforward, borrowing many elements from C and C++ but removing the complexities of pointers and multiple inheritance.
- **Secure**: Java provides several security features, including bytecode verification, secure class loading, and a security manager. These features help protect against common security threats such as viruses and malicious code.
- **Robust**: Java's robust nature comes from its strong memory management, exception handling, and type-checking mechanisms. These features help developers create reliable and error-free applications.
- **Multithreaded**: Java supports multithreading, allowing multiple threads of execution to run concurrently. This is particularly useful for developing interactive applications and improving performance.
- **High Performance**: While Java is an interpreted language, its performance is enhanced through Just-In-Time (JIT) compilation, which translates bytecode into native machine code at runtime.
- **Distributed**: Java has built-in support for developing distributed applications, making it

easy to create network-based applications that can communicate over the internet.

3) Understanding of JVM , JRE AND JDK :

- **JVM (Java Virtual Machine):** The JVM is an abstract machine that provides the runtime environment in which Java bytecode can be executed. It is responsible for converting bytecode into machine code that can be executed by the host system.
- **JRE (Java Runtime Environment)**: The JRE includes the JVM and a set of libraries and other components required to run Java applications. It does not include development tools like compilers and debuggers. The JRE is used to run Java programs on a user's machine.
- **JDK (Java Development Kit)**: The JDK is a complete software development kit that includes the JRE, as well as development tools such as the Java compiler (javac), the Java debugger, and other utilities. Developers use the JDK to create and compile Java programs.

4) Setting up the Java Environment and IDE :

- **Download and Install JDK:** Visit the Oracle JDK download page and download the latest version of the JDK for your operating system. Follow the installation instructions to install it on your machine.
- **Set Up Environment Variables:** Configure the JAVA_HOME environment variable to point to the JDK installation directory. Also, add the JDK's bin directory to the PATH environment variable to enable easy access to Java commands from the command line.
- **Install an IDE:** Choose an Integrated Development Environment (IDE) to write and manage your Java code. Popular IDEs for Java development include Eclipse, IntelliJ IDEA, and NetBeans. Download and install your preferred IDE.

5) Java Program Structure :

A.) Packages : A namespace for organizing classes and interfaces, helps avoid naming conflicts.

Example : package com.example;

B.) Classes : The blueprint from which individual objects are created.

Always write Class name in PascalCase

Example: public class Example

```
{
    // Class body
}
```

C.) Methods : A collection of statements

grouped together to perform an operation.

Always write method Name and fieldname in camelCase.

Example:

```
public void methodName()
{
    // Method body
 }
```

Example Code Structure:

```
package com.example;

public class Main
```

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

2.) Data Types, Variables, and Operators :

- **Theory:**

1) Primitive Data Types in Java :

Java servers several built-in data types known as primitive data types.

These types are not objects and represent the most basic forms of data:

- **Int :** A 32-bit signed integer with a range of 2,147,483,648 to 2,147,483,647. Typically used for storing whole numbers.

Example : int age = 25;

Float : A single-precision 32-bit floating point number. It is less precise than double but more memory e icient. Used for decimal numbers when less precision is acceptable.

Example : float price =

10.99f;

Char : A single 16-bit Unicode character. Useful for storing individual characters.

Example :

char grade = 'A';

boolean : Represents only two possible values : true or false.

Example : booleanisJavaFun = true;

Double : A double-precision 64-bit floating-point number. It provides greater precision compared to float and is commonly used for decimal numbers.

Example: double distance = 12345.6789;

Byte : An 8-bit signed integer. It has a range of -128 to 127.

Used in situations where memory optimization is crucial.

Example : byte b = 100;

Short : A 16-bit signed integer. With a range from -32,768 to 32,767, it occupies less memory than an int.

Example : short s = 10000;

- **Long :** A 64-bit signed integer. Useful for larger integer values.

  Example : long l = 123456789L;

2) Variable Declaration and Initialization :

variables must be declared before they can be used. A variable declaration specifies the type and name of the variable. Initialization assigns a value to the variable.

- **Declaration :** Specifies the variable type and name.

Example : int number;

- **Initialization :** Assigns a value to the declared variable.

  Example : number = 10;

- **Combined Declaration and Initialization :**

  Commonly used to simplify code.

  Example : int number = 10;

3) Operators :

Operators are special symbols that perform specific operations on one or more operands. They are classified as follows :

a.) Arithmetic Operators : Used for basic mathematical operations.

i. **Addition (+) :** Adds two operands.

   int sum = 5 + 3; // 8

ii. **Subtraction (-) :** Subtracts the second operand from the first.

   int dirence = 5 - 3; // 2

iii. **Multiplication (*) :** Multiplies two operands.

```
int product = 5 * 3; // 15
```

iv. **Division (/) :** Divides the first operand by the second.

```
int quotient = 5 / 3; // 1 (integer division)
```

v. **Modulus (%) :** Returns the remainder of the division of the first operand by the second.

```
int remainder = 5 % 3; // 2 .
```

b.) Relational Operators : Compare two values and return a boolean result.

Greater than (>) : Checks if the left operand is greater than the right.

```
boolean result = 5 > 3; // true
```

i. **Less than (<) :** Checks if the left operand is less than the right.

```
boolean result = 5 < 3; // false
```

ii. **Greater than or equal to (>=) :** Checks if the left operand is greater than or equal to the right.

boolean result = 5 >= 3; // true

iii. **Less than or equal to (<=) :** Checks if the left operand is less than or equal to the right.

boolean result = 5 <= 3; // false

iv. **Equal to (==) :** Checks if two operands are equal.

boolean result= 5 == 3; // false

v. **Not equal to (!=) :** Checks if two operands are not equal.

boolean result = 5 != 3; // true .

c.) Logical Operators : Perform logical operations and return a boolean result.

Logical AND (&&) : Returns true if both operands are true.

boolean result = true && false; // false

Logical OR (||) : Returns true if at least one operand is true.

boolean result = true || false; // true

Logical NOT (!) : Inverts the boolean value.

boolean result = !true; // false .

d.) **Assignment Operators :** Assign values to variables.

i. **Simple assignment (=) :** Assigns the value of the right operand to the left operand.

int x = 10;

ii. **Add and assign (+=) :** Adds the right operand to the left operand and assigns the result to the left operand.

x += 5; // x = x + 5

Subtract and assign (-=) : Subtracts the right operand from the left operand and assigns the result to the left operand.

x -= 5; // x = x - 5

Multiply and assign (*=) : Multiplies the left operand by the right operand and assigns the result to the left operand.

x *= 5; // x = x * 5

Divide and assign (/=) : Divides the left operand by the right operand and assigns the result to the left operand.

x /= 5; // x = x / 5

Modulus and assign (%=) : Takes the modulus of the left operand by the right operand and assigns the result to the left operand.

x %= 5; // x = x % 5 .

e.) Unary Operators : Operate on a single operand.

Unary plus (+) : Indicates a positive value (usually omitted).

int y = +5; // 5

Unary minus (-) : Negates a value.

int y = -5; // -5

Increment (++) : Increases the value of the operand by1.

x++; // Post-increment , ++x; // Pre-increment .

Decrement (--) : Decreases the value of the operand by 1.

x--; // Post-decrement

--x; // Pre-decrement

Logical complement (!) : Inverts the boolean value.

boolean z = !true; // false .

f.) Bitwise Operators : Perform operations on individual bits.

- **AND (&):** Performs a bitwise AND.

  int result = x &y;

- **OR (|):** Performs a bitwise OR.

  int result = x | y;

- **XOR (^):** Performs a bitwise XOR.

int result = x ^ y;

- **Complement (~) :** Inverts the bits.

int result = ~x;

- **Left shift (<<):** Shifts bits to the left.

int result = x <<2;

- **Right shift (>>):** Shifts bits to the right.

int result = x >>2;

- **Unsigned right shift (>>>):** Shifts bits to the right and fills leftmost bits with zero.

int result = x >>>2;

4) Type Conversion and Type Casting :

=> Java supports type conversion (automatic conversion) and type casting (explicit conversion) to help you work with di erent data types seamlessly.

- **Type Conversion :** The compiler automatically converts one data type to another, provided it is safe to do so.

Example : int x = 10; double y = x; // Automatic type conversion (int to double)

- **Type Casting :** Explicitly converting one data type to another using casting. This is necessary when you are converting from a larger type to a smaller type, or from a floating point type to an integer type.

Example : double a = 10.5; int b = (int) a; // Explicit type casting (double to int) .

3.) Control Flow Statements :

- **Theory :**

1) If-Else Statements :

⇨ The If-Else statement is a conditional control structure that allows you to execute different blocks of code based on the evaluation of a condition. It enables decision-making in your programs.

⇨ **Syntax :**

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

2) Switch Case Statements :

⇨ The Switch Case statement is a control structure that allows you to execute one block of code out of many based on the value of a variable. It is often used as an alternative to a series of If-Else statements when dealing with multiple conditions.

⇨ **Syntax :**

```
switch (expression) {
   case value1:
      // Code to execute if expression equals value1
      break;
   case value2:
      // Code to execute if expression equals value2
      break;
   // More cases...
   default:
      // Code to execute if none of the cases are matched
}
```

3) Loops (For, While, Do-While) :

- ⇨ Loops are used to execute a block of code repeatedly as long as a specified condition is met.
- ⇨ **For Loop**: The For loop is commonly used when the number of iterations is known beforehand.
- ⇨ **Syntax** :

```
for (initialization; condition; update) {
   // Code to execute repeatedly
}
```

⇨ **While Loop**: The While loop is used when the number of iterations is not known, and the loop depends on a condition being true.

⇨ **Syntax** :

```
while (condition) {
    // Code to execute repeatedly
}
```

⇨ **Do-While Loop**: The Do-While loop is similar to the While loop, but it guarantees that the loop body will be executed at least once.

⇨ **Syntax** :

```
do {
    // Code to execute repeatedly
} while (condition);
```

4) Break and Continue Keywords :

⇨ The Break and Continue keywords are used to control the flow of loops.

⇨ **Break**: The Break keyword is used to exit a loop or switch statement prematurely.

⇨ **Example** :

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i equals 5
```

```
        }
        System.out.println("Iteration: " + i);
    }
```

⇨ **Continue**: The Continue keyword is used to skip the current iteration of a loop and proceed with the next iteration.

⇨ **Example** :

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skip the iteration when i equals 5
    }
    System.out.println("Iteration: " + i);
}
```

4.) Classes and Objects :

- **Theory :**

1) Defining a Class and Object in Java :

=> Class : A class in Java is a blueprint for creating objects. It defines properties (fields) and behaviors (methods) that the objects created from the class will have.

Object : An object is an instance of a class. It is created from the class definition and can access the class's fields and methods.

2) Constructors and Overloading :

=> Constructor : A constructor is a special method used to initialize objects. It is called when an object is created and can set initial values for object fields.

Overloading : Constructor overloading is defining multiple constructors with different parameters in the same class.

Example :

```
public class Car {
    String color;
    int year;

    // Default constructor
    public Car() {
        color = "Unknown";
        year = 0;
    }
```

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

    void display() {
        System.out.println("Color: " + color + ", Year: "
+ year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car(); // Calls default
constructor
        Car car2 = new Car("Blue", 2022); // Calls
parameterized constructor
        car1.display();
        car2.display();
    }
```

```
}
```

3) Object Creation, Accessing Members of the Class :

=> Object Creation : Objects are created using the new keyword followed by the class constructor.

Accessing Members : Object members (fields and methods) are accessed using the dot (.) operator.

Example :

```
public class Car {
    String color;
    int year;

    void display() {
        System.out.println("Color: " + color + ", Year: " + year);
    }
}

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

```
        Car myCar = new Car(); // Object creation
        myCar.color = "Red"; // Accessing fields
        myCar.year = 2020; // Accessing fields
        myCar.display(); // Accessing method
    }
}
```

4) this Keyword :

=> this Keyword : The this keyword is a reference to the current object. It is used to refer to instance variables and methods of the current class. It is especially useful when parameter names are the same as instance variable names.

Example :

```
public class Car {
    String color;
    int year;

    // Parameterized constructor using 'this' keyword
    public Car(String color, int year) {
```

```
        this.color = color; // 'this.color' refers to the instance variable
        this.year = year; // 'this.year' refers to the instance variable
    }

    void display() {
        System.out.println("Color: " + color + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Green", 2021); // Object creation
        myCar.display(); // Accessing method
    }
}
```

5.) Methods in Java :

- **Theory :**

1) Defining Methods :

=> Method : A method is a block of code within a class that performs a specific task. It can be called or invoked to execute the defined actions.

=> Syntax :

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

=> **Example :**

```
public class MyClass
{
   void display()
   {
      System.out.println("Hello, World!");
   }
}
```

2) Method Parameters and Return Types :

=> Parameters : Methods can take parameters (also called arguments) to pass data. These parameters are defined within the parentheses in the method signature.

=> **Return Type :** The return type specifies the type of value the method returns. If the method does not return a value, the return type is void.

=> **Example :**

```
public class Calculator {
    // Method with parameters and return type
    int add(int a, int b) {
        return a + b;
    }

    // Void method
    void displayMessage(String message) {
        System.out.println(message);
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(5, 3); // Calling method
with parameters
        System.out.println("Sum: " + result);
        calc.displayMessage("Hello, Java!"); // Calling
void method
    }
}
```

3) Method Overloading :

=> **Method Overloading :** Method overloading allows a class to have more than one method with the same name, but different parameter lists (different types or numbers of parameters). It provides multiple ways to call a method based on different inputs.

=> **Example :**

```
public class Calculator {
   // Method overloading with different parameter counts
   int add(int a, int b) {
      return a + b;
   }

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

   // Method overloading with different parameter types
   double add(double a, double b) {
      return a + b;
   }
}

public class Main {
   public static void main(String[] args) {
      Calculator calc = new Calculator();
```

```
        System.out.println("Sum (2 int): " + calc.add(5,
3));
        System.out.println("Sum (3 int): " + calc.add(5,
3, 2));
        System.out.println("Sum (2 double): " +
calc.add(5.0, 3.5));
    }
}
```

4) Static Methods and Variables :

=> Static Methods : Static methods belong to the class rather than an instance of the class. They can be called without creating an instance of the class and can only access static data.

=> Static Variables : Static variables are shared among all instances of a class. They are also known as class variables.

=> Example :

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

    // Static method
    static int square(int number) {
```

```
        return number * number;
    }

    void incrementCount() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        // Accessing static method without creating an
instance
        System.out.println("Square of 4: " +
MathUtils.square(4));

        MathUtils obj1 = new MathUtils();
        MathUtils obj2 = new MathUtils();

        // Accessing and modifying static variable
        obj1.incrementCount();
        obj2.incrementCount();
```

```
        System.out.println("Count:          "          +
MathUtils.count);
    }
}
```

6.) Object Oriented Programming Concepts :

- **Theory :**

1) Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction :

=> A.) Encapsulation:

- **Definition:** Encapsulation is the concept of wrapping data (variables) and methods (functions) into a single unit called a class. It restricts direct access to some of an object's components and can prevent the accidental modification of data.
- Encapsulation is achieved through access modifiers like private, protected, and public which control the visibility of class members.

B.) Inheritance:

- **Definition:** Inheritance allows a new class (subclass or derived class) to inherit properties and behaviors (methods) from an existing class (superclass or base class).
- Inheritance promotes code reusability and establishes a natural hierarchy.

C.) Polymorphism:

- **Definition:** Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name. It can be achieved through method overloading and method overriding.
- Polymorphism enhances flexibility and maintainability of code.

D.) Abstraction:

- **Definition:** Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object.
- Abstraction is achieved using abstract classes and interfaces.

2) Inheritance: Single, Multilevel, Hierarchical :

=> A.) Single Inheritance : Single inheritance is when a class inherits from one superclass.

Example :

```
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
```

```
    }
}

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

B.) Multilevel Inheritance : Multilevel inheritance is when a class inherits from another subclass, forming a chain of inheritance.

Example :

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

class Puppy extends Dog {
    void weep() {
        System.out.println("The puppy weeps.");
```

```
    }
}
```

C.) Hierarchical Inheritance : Hierarchical inheritance is when multiple classes inherit from a single superclass.

Example :

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}
```

3) Method Overriding and Dynamic Method Dispatch :

=> **A.) Method Overriding :** Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

=> The method in the subclass should have the same name, return type, and parameters as in the superclass. It allows a subclass to offer a specific behavior.

=> **Example =**

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

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

B.) Dynamic Method Dispatch : Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile-time.

=> It is achieved using method overriding and upcasting.

=> **Example =**

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

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

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

```
    }
}
```

7.) Object Oriented Programming Concepts :

- **Theory :**

1) Constructor Types (Default, Parameterized) :

=> Default Constructor : default constructor is a constructor that takes no arguments. If no constructors are explicitly defined in a class, the Java compiler automatically provides a default constructor.

=> It initializes object fields with default values (e.g., null for objects, 0 for integers).

=> Example =

```
public class MyClass {
    int value;

    // Default constructor
    public MyClass() {
        value = 0; // Initializing value
    }
}
```

=> **Parameterized Constructor** : A parameterized constructor is a constructor that takes arguments to initialize object fields with specific values at the time of object creation.
=> It allows initializing an object with user-defined values.
=> **Example =**

```
public class MyClass {
   int value;

   // Parameterized constructor
   public MyClass(int val) {
      value = val;
   }
}
```

2) Copy Constructor (Emulated in Java) :
=> Java does not support a built-in copy constructor like C++. However, it can be emulated by defining a constructor that takes an object of the same class and copies its fields.
=> Used to create a new object as a copy of an existing object.
=> Example =

```
public class MyClass {
   int value;

   // Parameterized constructor
```

```
    public MyClass(int val) {
        value = val;
    }

    // Copy constructor
    public MyClass(MyClass obj) {
        this.value = obj.value;
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass original = new MyClass(10);
        MyClass copy = new MyClass(original); // Creating a copy
        System.out.println("Original value: " + original.value);
        System.out.println("Copy value: " + copy.value);
    }
}
```

3) Constructor Overloading :

=> Constructor overloading is the process of having more than one constructor in a class, each with different parameter lists. It allows different ways to initialize objects.

=> Provides flexibility in object creation.

=> Example =

```
public class MyClass {
    int value;
    String name;

    // Default constructor
    public MyClass() {
        value = 0;
        name = "Unknown";
    }

    // Parameterized constructor
    public MyClass(int val) {
        value = val;
        name = "Unknown";
    }

    // Parameterized constructor with two parameters
    public MyClass(int val, String name) {
        value = val;
        this.name = name;
    }
}

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

```
        MyClass obj1 = new MyClass(); // Calls default constructor
        MyClass obj2 = new MyClass(10); // Calls parameterized constructor
        MyClass obj3 = new MyClass(20, "Java"); // Calls parameterized constructor with two parameters
    }
}
```

4) Object Life Cycle and Garbage Collection :

=> A.) Object Life Cycle:

→ The object life cycle in Java refers to the stages an object goes through from its creation to its destruction.

→ **Stages:**

- ➔ **Creation:** An object is created using the new keyword, which allocates memory and calls the constructor.
- ➔ **Usage:** The object can be used, its methods can be called, and its fields can be accessed or modified.
- ➔ **Destruction:** When an object is no longer referenced, it becomes eligible for garbage collection.

B.) Garbage Collection:

→ Garbage collection is an automatic process in Java that reclaims memory occupied by objects that are no longer in use, preventing memory leaks.

→ The JVM's garbage collector runs periodically to identify and remove unreferenced objects.

→ Developers can suggest garbage collection using System.gc(), but it is not guaranteed to run immediately.

→ Finalization (finalize method) is called before the garbage collector removes an object, but it is generally discouraged to rely on it for critical operations.

=> **Example :**

```
public class MyClass {
   @Override
   protected void finalize() throws Throwable {
      System.out.println("Object is being garbage collected");
   }
}

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

```
        MyClass obj = new MyClass();
        obj = null; // Dereference the object
        System.gc(); // Suggest garbage collection
    }
}
```

8.) Arrays and Strings :

- **Theory :**

1) One-Dimensional and Multidimensional Array :

=> A.) One-Dimensional Arrays:

→ A one-dimensional array is a list of variables of the same type, accessed by a single index.

→ **Syntax :**

```
int[] array = new int[10];
```

→ **Example :**

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
System.out.println(numbers[i]);
    }
```

B.) Multi-Dimensional Arrays:

→ Multidimensional arrays are arrays of arrays. A two-dimensional array is the most common form, resembling a matrix.

→ **Syntax :**

```
int[][] array = new int[3][4];
```

→ **Exmaple :**

```
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();
  }
```

2) String Handling in Java: String Class, StringBuffer, StringBuilder:

=> A) String Class:

→ The String class is immutable, meaning once a String object is created, its value cannot be changed**.**

→ **Example :**

```
String str = "Hello";
String newStr = str.concat(" World");
System.out.println(newStr); // Output: Hello World
```

B) StringBuffer:

→ StringBuffer is mutable, meaning it can be modified after creation. It is thread-safe.

→ **Example :**

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

C) StringBuilder:

→ StringBuilder is also mutable but is not thread-safe. It is faster than StringBuffer for single-threaded applications.

→ **Example :**

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

3) Array of Objects :

=> An array of objects is an array where each element is an object.

=> Example :

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

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

```
public class Main {
    public static void main(String[] args) {
        Student[] students = new Student[3];
        students[0] = new Student("Alice", 20);
        students[1] = new Student("Bob", 22);
        students[2] = new Student("Charlie", 23);

        for (Student s : students) {
            System.out.println(s.name + " is " + s.age + "
years old.");
        }
    }
}
```

4) String Methods (length, charAt, substring, etc.)

=> A) length():

→ Returns the length of the string.

→ **Example :**

```
String str = "Hello";

int len = str.length();

System.out.println(len); // Output: 5
```

B) charAt(int index):

→Returns the character at the specified index.

→ **Example :**

```
String str = "Hello";
char ch = str.charAt(1);
System.out.println(ch); // Output: e
```

C) substring(int beginIndex):

→ Returns a new string that is a substring of the original string, starting from the specified index.

→ **Example :**

```
String str = "Hello";
String sub = str.substring(2);
System.out.println(sub); // Output: llo
```

D) substring(int beginIndex, int endIndex):

→ Returns a new string that is a substring of the original string, starting from beginIndex and ending at endIndex.

→ **Example :**

```
String str = "Hello";
String sub = str.substring(1, 4);
System.out.println(sub); // Output: ell
```

9.) Inheritance and Polymorphism :

- **Theory :**

1) Inheritance Types and Benefits :

=> A.) Single Inheritance:

→ A class inherits from one superclass.

→ Simplifies the inheritance hierarchy.

→ **Example :**

```
class Animal {
void eat() {
System.out.println("This animal eats food.");
}
}

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

B.) Multilevel Inheritance:

→ A class inherits from another derived class, forming a chain.

→ Allows a deeper inheritance hierarchy.

→ **Example :**

```
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
```

```
    }
}

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

class Puppy extends Dog {
    void weep() {
        System.out.println("The puppy weeps.");
    }
}
```

C.) Hierarchical Inheritance:

→Multiple classes inherit from one superclass.

→Promotes reusability and modular design.

→**Example :**

```
class Animal {
    void eat() {
        System.out.println("This animal eats
food.");
    }
}

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

```
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}
```

→ **Benefits :**

Code Reusability: Reuse existing code in new contexts.

Simplified Maintenance: Fixes in base class automatically apply to derived classes.

Polymorphism: Achieve flexibility by using a superclass reference to access subclass objects.

Logical Hierarchy: Provides a natural hierarchical classification.

2) Method Overriding :

→ Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

→ The overridden method must have the same name, return type, and parameters. It is used to achieve runtime polymorphism.

→ **Example :**

```
class Animal {
```

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

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

3) Dynamic Binding (Run-Time Polymorphism) :

=> Dynamic binding, also known as run-time polymorphism, occurs when the method to be invoked is determined at runtime rather than compile-time.

→ It is achieved through method overriding and the use of a superclass reference to refer to a subclass object.

→ **Example:**

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

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

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

4) Super Keyword and Method Hiding :

=> A) Super Keyword:

→ The super keyword is used to refer to the immediate parent class object. It can be used to access parent class members and constructors.

→ Access Parent Class Constructor: Call parent class constructor from subclass constructor.

→ Access Parent Class Methods: Call overridden methods from parent class.

→ **Example :**

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

class Dog extends Animal {
    @Override
    void sound() {
        super.sound(); // Calls parent class method
        System.out.println("Dog barks");
    }
}
```

B) Method Hiding:

→ Method hiding occurs when a static method in a subclass has the same name and parameter list as a static method in its superclass.

→ Unlike method overriding, where the method to be called is determined at runtime, in method hiding, the method to be called is determined at compile-time.

→ **Example :**

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

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

public class Test {
    public static void main(String[] args) {
        Animal.staticMethod(); // Calls static method in Animal
        Dog.staticMethod(); // Calls static method in Dog
    }
}
```

10.) Interfaces and Abstract Class :

- **Theory :**

1) Abstract Classes and Methods :

=> A) Abstract Class:

→ An abstract class is a class that cannot be instantiated on its own and must be subclassed. It can contain abstract methods (without a body) and concrete methods (with a body).

→ Used to define a common template for its subclasses.

→ Contains at least one abstract method.

=> Example =

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

void eat() { // Concrete method
System.out.println("This animal eats food.");
}
}
```

B) Abstract Method:

→ An abstract method is a method that is declared without an implementation and must be implemented by subclasses.

→ Ensures that all subclasses provide a specific implementation for the method.

=> **Example =**

```
abstract class Animal {
```

```
    abstract void sound(); // Abstract method
}

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

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

2) Interfaces: Multiple Inheritance in Java :

=> A) Interface:

→ An interface is a reference type in Java that can contain abstract methods, default methods, static methods, and constants. Interfaces provide a way to achieve multiple inheritance in Java.

→ A class can implement multiple interfaces.

→ Interfaces specify what a class must do but not how it does it.

=> Syntax :

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

B) Multiple Inheritance using Interfaces:

→ Multiple inheritance is achieved in Java by implementing multiple interfaces. A class can implement more than one interface, thus inheriting the abstract methods of all the interfaces.

→ This allows a class to combine behaviors defined in different interfaces.

→ **Example =**

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

interface Pet {
   void play();
}

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

   @Override
```

```
    public void play() {
        System.out.println("Dog plays");
    }
}
```

3) Implementing Multiple Interfaces :

=> A class can implement multiple interfaces by providing implementations for all the abstract methods declared in those interfaces.

→ The class must provide concrete implementations for all the methods from all the interfaces it implements.

→ This allows the class to adhere to multiple contracts and behaviors.

=> Example :

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

interface Showable {
    void show();
}
```

```
class Document implements Printable, Showable {
    @Override
    public void print() {
        System.out.println("Printing document");
    }

    @Override
    public void show() {
        System.out.println("Showing document");
    }
}

public class Main {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print(); // Calls Printable's print method
        doc.show();  // Calls Showable's show method
    }
}
```

11.) Packages and Access Modifiers :

- **Theory :**

1) Java Packages: Built-in and User-Defined Packages :

=> A) Built-in Packages:

→ Built-in packages are provided by the Java API and contain predefined classes and interfaces that developers can use directly.

→ **Examples:**

⇨ java.lang (automatically imported, contains fundamental classes like String, Math, Integer, etc.).

⇨ java.util (contains utility classes like ArrayList, HashMap, Date, etc.).

⇨ java.io (contains classes for input and output operations like File, InputStream, OutputStream, etc.).

import java.util.ArrayList;

=> **Example :**

```
public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        System.out.println(list);
    }
}
```

B) User-Defined Packages:

→ User-defined packages are created by developers to group their own related classes and interfaces, promoting better organization and modularity of the code.

→ **Creating a Package :**

```
package com.example.myapp;

public class MyClass {
   public void display() {
      System.out.println("Hello from MyClass in
com.example.myapp package!");
   }
}
```

→ **Using a User-Defined Package** :

```
import com.example.myapp.MyClass;

public class Main {
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      obj.display();
   }
}
```

2) Access Modifiers: Private, Default, Protected, Public :

=> A) Private :

=> The member is accessible only within the same class. It is the most restrictive access level.

=> Example :

```
public class MyClass {
private int value;

private void display() {
System.out.println("Value: " + value);
  }
}
```

B) Default (no modifier):

=> The member is accessible only within classes in the same package. It is more restrictive than protected but less restrictive than private.

=> Example :

```
public class MyClass {
   int value; // Default access

   void display() {
      System.out.println("Value: " + value);
   }
}
```

C) Protected :

=> The member is accessible within the same package and by subclasses even if they are in different packages.

=> Example :

```
public class MyClass {
   protected int value;

   protected void display() {
      System.out.println("Value: " + value);
   }
}
```

D) Public:

=> The member is accessible from any other class.

=> Example :

```
   public class MyClass {
  public int value;

  public void display() {
     System.out.println("Value: " + value);
  }
}
```

3) Importing Packages and Classpath :

=> A) Importing Packages :

→ To use classes and interfaces from a different package, the import statement is used.

→ **Syntax :**

import package_name.ClassName;

import package_name.*;

→ **Example :**

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
System.out.println(list);
   }
 }
```

B) Classpath :

→ The classpath is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to look for user-defined classes and packages.

→ **Setting The Path :**

java -cp /path/to/classes MainClass

→ **Environment Variable :**

set CLASSPATH=/path/to/classes

12.) Exception Handling :

- **Theory :**

1) Types of Exceptions: Checked and Unchecked :

=> A) Checked Exceptions :

→ Checked exceptions are exceptions that are checked at compile-time. These must be either caught or declared in the method signature using the throws keyword.

→ **Example :** IOException, SQLException, FileNotFoundException.

→ Ensures that the programmer handles the potential errors, making the code more robust.

B) Unchecked Exceptions :

→ Unchecked exceptions are exceptions that are not checked at compile-time but at runtime. They are subclasses of RuntimeException.

→**Example :** NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

→Typically indicate programming errors, such as logic errors or improper use of APIs.

2) try, catch, finally, throw, throws :

=> A) try Block :

→ The try block contains code that might throw an exception. It must be followed by either a catch block or a finally block.

→ **Syntax :**

```
try {
// Code that may throw an exception
}
```

B) catch Block:

→ The catch block handles the exception thrown by the try block. It must specify the type of exception it can handle.

→ **Syntax :**

```
try {
  // Code that may throw an exception
} catch (ExceptionType e) {
  // Code to handle the exception
}
```

C) finally Block:

→ The finally block contains code that is always executed, regardless of whether an exception is thrown or not. It is typically used for cleanup code, like closing files or releasing resources.

→ **Syntax :**

```
try {
  // Code that may throw an exception
} catch (ExceptionType e) {
  // Code to handle the exception
} finally {
  // Code that will always execute
```

}

D) throw Keyword:

→ The throw keyword is used to explicitly throw an exception from a method or any block of code.

→ **Syntax :**

```
throw new ExceptionType("Error message");
```

E) throws Keyword:

→ The throws keyword is used in a method signature to declare that a method might throw one or more exceptions. This informs the caller of the method to handle or declare these exceptions.

→ **Syntax :**

```
public void myMethod() throws ExceptionType1, ExceptionType2 {
  // Method code
}
```

3) Custom Exception Classes :

=> Custom exception classes are user-defined exceptions that extend Exception (for checked exceptions) or RuntimeException (for unchecked exceptions). They allow developers to create exceptions that are specific to their application's requirements.

=> Syntax :

```
class MyCustomException extends Exception {
```

```
    public MyCustomException(String message) {
        super(message);
    }
}
```

=> Example :

```
        class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}
public class Main {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (MyCustomException e) {
            System.out.println("Caught an exception: " +
e.getMessage());
        }
    }
```

```
    static void validateAge(int age) throws MyCustomException {
        if (age < 18) {
            throw new MyCustomException("Age must be 18 or older.");
        } else {
            System.out.println("Age is valid.");
        }
    }
}
```

13.) Multi-threading :

- **Theory :**

1) Introduction to Threads :

=> A thread is a lightweight sub-process, a smallest unit of processing. It is a separate path of execution. Threads allow multiple operations to run concurrently within a single process, improving the efficiency and performance of applications.

2) Creating Threads by Extending Thread Class or Implementing Runnable Interface :

=> **A) Extending Thread Class:**

→ A class can extend the Thread class and override its run() method to define the code that should be executed by the thread.

→ **Example :**

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

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

B) Implementing Runnable Interface :

→ A class can implement the Runnable interface and override its run() method to define the code that should be executed by the thread. The Runnable interface provides a way to create threads without subclassing the Thread class.

→ **Example :**

```
    class MyRunnable implements Runnable {
```

```
    public void run() {
        System.out.println("Thread is running by
implementing Runnable interface.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new
MyRunnable();
        Thread thread = new
Thread(myRunnable);
        thread.start(); // Starts the thread
    }
}
```

3) Thread Life Cycle :

=> The life cycle of a thread in Java includes several states:

1. **New:** A thread is in the new state if it is created but not yet started.
2. **Runnable:** After calling the start() method, the thread is in the runnable state.
3. **Blocked:** A thread is blocked when it is waiting for a monitor lock to enter a synchronized block/method.

4. **Waiting:** A thread is in the waiting state if it is waiting indefinitely for another thread to perform a particular action.
5. **Timed Waiting:** A thread is in the timed waiting state if it is waiting for a specified amount of time.
6. **Terminated:** A thread is in the terminated state once it exits the run() method.

→ **Example :**

```
public class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println(i);
                Thread.sleep(1000); // Thread goes to
timed waiting state
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Thread moves to runnable
state
    }
}
```

4) Synchronization and Inter-thread Communication :

=> A) Synchronization :

→ Synchronization is the capability to control the access of multiple threads to shared resources. It prevents thread interference and ensures consistency.

→ **Example :**

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

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

   public int getCount() {
      return count;
   }
}
```

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

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();
```

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

B) Inter-thread Communication:

→ Inter-thread communication in Java is used to avoid thread polling and allows threads to communicate with each other using methods like wait(), notify(), and notifyAll().

→ **Example :**

```
class SharedResource {
  private int count = 0;
  private boolean available = false;

  public synchronized void produce() throws
InterruptedException {
    while (available) {
      wait();
    }
    count++;
```

```
        System.out.println("Produced: " + count);
        available = true;
        notify();
    }

    public synchronized void consume() throws InterruptedException {
        while (!available) {
            wait();
        }
        System.out.println("Consumed: " + count);
        available = false;
        notify();
    }
}

public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
```

```
Runnable producerTask = () -> {
    try {
        for (int i = 0; i < 5; i++) {
            resource.produce();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
};

Runnable consumerTask = () -> {
    try {
        for (int i = 0; i < 5; i++) {
            resource.consume();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
};
```

```
        Thread producer = new Thread(producerTask);
        Thread consumer = new Thread(consumerTask);

        producer.start();
        consumer.start();
    }
}
```

14.) File Handling :

- **Theory :**

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

=> Java provides a comprehensive java.io package for performing input and output (I/O) operations on files. This package contains classes and interfaces for reading from and writing to files.

2) FileReader and FileWriter Classes :

=> A) FileReader:

→ The FileReader class is used to read data from files in the form of characters. It is a subclass of InputStreamReader.

→ Typically used for reading text files.

→ **Example :**

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

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

B) File Writer:

→ The FileWriter class is used to write data to files in the form of characters. It is a subclass of OutputStreamWriter.

→ Typically used for writing text files.

→ **Example :**

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

```
import java.io.IOException;

public class Main {
   public static void main(String[] args) {
      try {
         FileWriter writer = new
FileWriter("example.txt");
         writer.write("Hello, World!");
         writer.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

3) BufferedReader and BufferedWriter :

=> A) BufferedReader:

→ The BufferedReader class reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines.

→ Often used to read text from a file line by line.

→ **Example :**

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

```
public class Main {
   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();
      }
   }
}
```

B) BufferedWriter:

=> The BufferedWriter class writes text to a character-output stream, buffering characters to provide efficient writing of characters, arrays, and lines.

→ Often used to write text to a file line by line.

→ **Example :**

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

```
public class Main {
   public static void main(String[] args) {
      try {
         BufferedWriter writer = new
BufferedWriter(new FileWriter("example.txt"));
         writer.write("Hello, World!");
         writer.newLine();
         writer.write("Welcome to Java File I/O.");
         writer.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

4) Serialization and Deserialization :

=> A) Serialization:

→ Serialization is the process of converting an object's state into a byte stream, so it can be saved to a file or transmitted over a network.

→ Classes must implement the Serializable interface to be serialized.

→ **Example =**

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

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

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;

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

public class Main {
    public static void main(String[] args) {
        try {
            Student student = new Student("Alice",
20);
```

```
            FileOutputStream fileOut = new
        FileOutputStream("student.ser");
            ObjectOutputStream out = new
        ObjectOutputStream(fileOut);
            out.writeObject(student);
            out.close();
            fileOut.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

B) Deserialization:

=> Deserialization is the process of converting a byte stream back into a copy of the original object.

→ The byte stream must be a valid serialization stream of a class that implements Serializable.

→ **Example :**

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

```
import java.io.ObjectInputStream;


public class Main {
   public static void main(String[] args) {
      try {
         FileInputStream fileIn = new FileInputStream("student.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         Student student = (Student) in.readObject();
         in.close();
         fileIn.close();
         System.out.println("Name: " + student.name);
         System.out.println("Age: " + student.age);
      } catch (IOException | ClassNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

15.) Collection Framework :

- **Theory :**

1) Introduction to Collections Framework :

=> Java Collections Framework provides a set of classes and interfaces that implement commonly reusable collection data structures. This framework provides a standard way to handle groups of objects.

2) List, Set, Map, and Queue Interfaces :

=> A) List Interface :

→ An ordered collection (also known as a sequence). Lists can contain duplicate elements and allow positional access and insertion of elements.

→**Implementations**: ArrayList, LinkedList, Vector, Stack.

→ **Example :** List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");

B) Set Interface :

→ A collection that does not allow duplicate elements. Sets are unordered and do not support positional access.

→**Implementations** : HashSet, TreeSet, LinkedHashSet.

→ **Example :**

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
```

C) Map Interface :

→ A collection of key-value pairs. Each key can map to at most one value. Maps do not allow duplicate keys.

→**Implementations** : HashMap, TreeMap, LinkedHashMap, Hashtable.

→ **Example :**

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Apple");
map.put(2, "Banana");
map.put(3, "Cherry");
```

D) Queue Interface :

→ A collection used to hold multiple elements prior to processing. Typically, queues order elements in a FIFO (first-in, first-out) manner.

→ **Implementations** : LinkedList, PriorityQueue, ArrayDeque.

→ **Example :**

```
Queue<String> queue = new LinkedList<>();
```

```
queue.add("Apple");
queue.add("Banana");
queue.add("Cherry");
```

3) ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap.

=> A) ArrayList :

→ A resizable array implementation of the List interface. Provides fast random access to elements.

→ **Example :**

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Apple");
arrayList.add("Banana");
arrayList.add("Cherry");
```

B) LinkedList :

→ A doubly linked list implementation of the List and Deque interfaces. Provides better performance for insertions and deletions compared to ArrayList.

→ **Example :**

```
List<String> linkedList = new LinkedList<>();
linkedList.add("Apple");
linkedList.add("Banana");
linkedList.add("Cherry");
```

C) HashSet :

→ A hash table-backed implementation of the Set interface. Does not guarantee any order of elements.

→ **Example :**

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");
hashSet.add("Cherry");
```

D) TreeSet:

→ A red-black tree-based implementation of the NavigableSet interface. Maintains elements in a sorted order.

→ **Example :**

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Apple");
treeSet.add("Banana");
treeSet.add("Cherry");
```

E) HashMap:

→ A hash table-backed implementation of the Map interface. Allows null values and one null key. Does not guarantee any order of entries.

→ **Example :**

```
Map<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "Apple");
hashMap.put(2, "Banana");
```

hashMap.put(3, "Cherry");

F) TreeMap:

→ A red-black tree-based implementation of the NavigableMap interface. Maintains keys in a sorted order.

→ **Example :**

```
Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(1, "Apple");
treeMap.put(2, "Banana");
treeMap.put(3, "Cherry");
```

4) Iterators and ListIterators :

=> A) Iterator :

→ An interface used to iterate over collections, such as List, Set, and Map. Provides methods to check the next element, retrieve it, and remove it**.-**

→ **Important Methods:**

→ **hasNext():** Returns true if the iteration has more elements.

→ **next():** Returns the next element in the iteration.

→ **remove():** Removes the last element returned by the iterator.

→ **Example :**

```
List<String> list = new ArrayList<>();
```

```
list.add("Apple");
list.add("Banana");
list.add("Cherry");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

B) ListIterators :

→ An interface that extends Iterator and is specific to lists. Provides additional methods to traverse the list in both directions and modify the list during iteration.

→ **Important Methods**:

→ **hasPrevious():** Returns true if there are previous elements.

→ **previous():** Returns the previous element in the list.

→ **add(E e):** Inserts the specified element into the list.

→ **Example :**

```
List<String> list = new ArrayList<>();
list.add("Apple");
```

```
list.add("Banana");
list.add("Cherry");

ListIterator<String> listIterator = list.listIterator();
while (listIterator.hasNext()) {
   System.out.println(listIterator.next());
}

while (listIterator.hasPrevious()) {
   System.out.println(listIterator.previous());
}
```

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

- **Theory :**

1) Streams in Java (InputStream, OutputStream) :

=> A) InputStream:

→ InputStream is an abstract class that represents a stream of bytes for reading data. It is the superclass for all classes representing an input stream of bytes.

→ **Important Subclasses**: FileInputStream, ByteArrayInputStream, FilterInputStream.

→ **Example :**

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

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

B) OutputStream :

→ OutputStream is an abstract class that represents a stream of bytes for writing data. It is the superclass for all classes representing an output stream of bytes.

→ **Important Subclasses**: FileOutputStream, ByteArrayOutputStream, FilterOutputStream.

→ **Example :**

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

public class Main {
public static void main(String[] args) {
try (FileOutputStream outputStream = new
FileOutputStream("example.txt")) {
String str = "Hello, World!";
outputStream.write(str.getBytes());
} catch (IOException e) {
e.printStackTrace();
 }
}
}
```

2) Reading and Writing Data Using Streams :

=> A) Reading Data:

→ Reading data from a stream involves retrieving bytes of data from an input source, such as a file.

-> Steps:

1. Create an InputStream object.
2. Read bytes using read() method.
3. Handle exceptions.
4. Close the stream to release resources.

→ **Example :**

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

```
import java.io.IOException;

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

B) Writing Data:

→ Writing data to a stream involves sending bytes of data to an output destination, such as a file.

-> Steps:

1. Create an OutputStream object.
2. Write bytes using write() method.
3. Handle exceptions.
4. Close the stream to release resources.

→ **Example :**

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

```
public class Main {
   public static void main(String[] args) {
      try (FileOutputStream outputStream = new
FileOutputStream("example.txt")) {
         String str = "Hello, World!";
         outputStream.write(str.getBytes());
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

3) Handling File I/O Operations :

=> A) FileReader and FileWriter:

(i) FileReader: Used to read characters from a file.

→ Example :

```
import java.io.FileReader;

import java.io.IOException;

public class Main {

  public static void main(String[] args) {

                try (FileReader reader = new
FileReader("example.txt")) {

   int character;

   while ((character = reader.read()) != -1) {
```

```
            System.out.print((char) character);
            }
            } catch (IOException e) {
                  e.printStackTrace();
            }
            }
      }
```

(ii) FileWriter: Used to write characters to a file.

→ **Example :**

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


public class Main {
   public static void main(String[] args) {
      try (FileWriter writer = new
FileWriter("example.txt")) {
         writer.write("Hello, World!");
      } catch (IOException e) {
         e.printStackTrace();
      }
```

```
    }
}
```

B) BufferedReader and BufferedWriter:

(i) **BufferedReader**: Used to read text from an input stream, buffering characters for efficient reading.

→ **Example :**

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

public class Main {
    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);
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

(ii) **BufferedWriter**: Used to write text to an output stream, buffering characters for efficient writing.

→ **Example :**

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

public class Main {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt"))) {
            writer.write("Hello, World!");
            writer.newLine();
            writer.write("Welcome to Java File I/O.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

C) Serialization and Deserialization:

=> (i) **Serialization**: Converting an object's state into a byte stream.

--> **Example :**

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

class Student implements Serializable {
private static final long serialVersionUID = 1L;
String name;
int age;

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

```
}

public class Main {
public static void main(String[] args) {
try {
   Student student = new Student("Alice", 20);
        FileOutputStream fileOut = new FileOutputStream("student.ser");
   ObjectOutputStream out = new ObjectOutputStream(fileOut);
   out.writeObject(student);
   out.close();
   fileOut.close();
} catch (IOException e) {
   e.printStackTrace();
}
}
}
```

(ii) **Deserialization**: Converting a byte stream back into an object's state.

→ **Example :**

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


public class Main {
public static void main(String[] args) {
try {
        FileInputStream fileIn = new FileInputStream("student.ser");
   ObjectInputStream in = new ObjectInputStream(fileIn);
   Student student = (Student) in.readObject();
   in.close();
   fileIn.close();
   System.out.println("Name: " + student.name);
   System.out.println("Age: " + student.age);
} catch (IOException | ClassNotFoundException e) {
   e.printStackTrace();
}
}
```

}