JAVA – Theory(Module 1 – Core Java)

1. Introduction to Java Theory

1. History of Java

- **Developed by Sun Microsystems**: Java was developed by Sun Microsystems in 1995.
- **Creator**: James Gosling is known as the father of Java.
- **Initial Purpose**: Originally designed for interactive television, but too advanced for the digital cable television industry at the time.
- **Acquisition**: Oracle Corporation acquired Sun Microsystems in 2010.

2. Features of Java

- **Platform Independent**: Java code can run on any device that has a Java Virtual Machine (JVM), making it highly portable.
- **Object -Oriented**: Java is based on objects and classes, promoting reusability and modularity.
- **Simple**: Java has a clean syntax and easy-to-understand structure, making it simpler to learn and use.
- **Secure**: Java provides a secure environment for developing applications through features like bytecode verification and runtime security checks.
- **Multithreaded**: Java supports multithreading, allowing multiple threads to run simultaneously for efficient performance.
- **Robust**: Java emphasizes early checking for possible errors, using strong memory management and exception handling.
- **High Performance**: Java's Just-In-Time (JIT) compiler helps in

optimizing performance.

- **Distributed**: Java supports distributed computing through RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).

3. Understanding JVM, JRE, and JDK

- **JVM (Java Virtual Machine)**: An abstract machine that enables your computer to run a Java program. Converts bytecode into machine code.
- **JRE (Java Runtime Environment)**: Provides libraries, JVM, and other components to run Java applications. Does not include development tools like compilers.
- **JDK (Java Development Kit)**: A full-featured software development kit that includes JRE, compilers, and tools (debuggers, profilers) necessary for developing Java applications.

4. Setting up the Java Environment and IDE

- **Download and Install JDK**: Visit the Oracle website to download and install the latest JDK.
- **Set Environment Variables**: Configure JAVA_HOME and add bin directory to the PATH variable.
- **IDE Setup**:
    - **Eclipse**:
        - Download from the Eclipse website.
        - Install and configure workspace settings.
    - **IntelliJ IDEA**:
        - Download from the JetBrains website.

- Install and configure project settings.

5. Java Program Structure

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

```
package com.example;
```

- **Classes**: The blueprint from which individual objects are created.

```
public class Example {
    // Class body
}
```

- **Methods**: A collection of statements grouped together to perform an operation.

```
    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!");
    }
}
```

JAVA – Theory(Module 1 – Core Java)

2. Data Types, Variables, and Operators Theory

1. Primitive Data Types in Java

Java offers 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.

```
int age = 25;
```

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

```
float price = 10.99f;
```

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

```
char grade = 'A';
```

- **boolean**: Represents only two possible values: true or false. Commonly used in control structures.

```
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.

```
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.

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.

short s = 10000;

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

long l = 123456789L;

2. Variable Declaration and Initialization

In Java, 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.

int number;

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

number = 10;

- **Combined Declaration and Initialization**: Commonly used to simplify code.

int number = 10;

3. Operators

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

- **Arithmetic Operators**: Used for basic mathematical operations.
    - Addition (+): Adds two operands.

int sum = 5 + 3; // 8

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

int difference = 5 - 3; // 2

- Multiplication (*): Multiplies two operands.

int product = 5 * 3; // 15

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

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

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

int remainder = 5 % 3; // 2

- **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

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

boolean result = 5 < 3; // false

- Greater than or equal to (>=): Checks if the left operand is great er than or equal to the right.

boolean result = 5 >= 3; // true

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

boolean result = 5 <= 3; // false

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

boolean result = 5 == 3; // false

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

boolean result = 5 != 3; // true

- **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

- **Assignment Operators**: Assign values to variables.
    - Simple assignment (=): Assigns the value of the right operand to the left operand.

int x = 10;

    - 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

- **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 by 1.

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

- **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 different data types seamlessly.

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

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.

double a = 10.5;

int b = (int) a; // Explicit type casting (double to int)

JAVA – Theory(Module 1 – Core Java)

3. Control Flow Statements Theory

1. If-Else Statements

- **What**: The if-else statement is used to execute a block of code conditionally. If the condition evaluates to true, the if block is execute d; otherwise, the else block is executed.
- **Why**: This helps in making decisions in your code based on certain conditions.
- **When**: Use if-else statements when you need to perform different actions based on different conditions.
- **Theory**: The if-else statement is fundamental for controlling the flow of execution in a program. It allows the program to choose different paths based on conditions, which is essential for implementing logic.
- **Syntax**:

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

- **Example**:

```
int age = 18;
if (age >= 18) {
System.out.println("You are an adult.");
} else {
System.out.println("You are not an adult.");
```

}

2. Switch Case Statements

- **What**: The switch statement is used to execute one block of code from multiple conditions. Each condition is known as a case, and the value of the variable is compared with each case.
- **Why**: It simplifies the process of selecting one of many code blocks to be executed, making the code cleaner and more readable.
- **When**: Use switch statements when you need to compare a variable against multiple values and execute different blocks of code based on those values.
- **Theory**: The switch statement is a control statement that handles multiple selections and enumerations by passing control to one of the case statements within its body. It improves readability and efficiency when dealing with numerous conditions that depend on the value of a single variable.
- **Syntax**:

```
switch (variable) {
    case value1:
        // Code to execute if variable equals value1
break;
    case value2:
        // Code to execute if variable equals value2
break;
    // More cases...
```

```
    default:
        // Code to execute if variable doesn't match any case
}
```

- **Example**:

```
int day = 2;
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");
break;
}
```

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

- **What**: Loops are used to execute a block of code repeatedly as long as the condition is true.
- **Why**: They allow you to run the same block of code multiple times, which is useful for tasks that require repetition.

- **When**: Use loops when you need to perform an action repeatedly based on a condition or for a specific number of times.
- **Theory**: Loops are fundamental in programming for performing repetitive tasks efficiently. They reduce redundancy, simplify code, and ensure that the program can handle repeated operations without manual intervention.
- **For Loop**:
    - **Syntax**:

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

    - **Example**:

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

- **While Loop**:
    - **Syntax**:

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

    - **Example**:

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

```
i++;
}
```

- **Do-While Loop**:
  - **Syntax**:

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

  - **Example**:

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

4. Break and Continue Keywords

- **What**: These keywords are used to control the flow of loops.
- **Why**: They help manage loop execution by providing control over which iterations to execute or skip.
- **When**: Use break to exit a loop prematurely, and continue to skip the current iteration and proceed with the next one.
- **Theory**: The break statement exits the loop or switch statement, while the continue statement skips the current iteration and moves to the next one. These control statements provide finer control over loop execution, ensuring that the program can respond to specific conditions efficiently.

- **Break**: Terminates the loop immediately.
    - **Example**:

```
for (int i = 0; i< 10; i++) {
   if (i == 5) {
break;
   }
System.out.println(i);
}
```

- **Continue**: Skips the current iteration and proceeds with the next iteration.
    - **Example**:

```
for (int i = 0; i< 10; i++) {
   if (i == 5) {
continue;
   }
System.out.println(i);
}
```

JAVA – Theory(Module 1 – Core Java)

4. Classes and Objects Theory

1. Defining a Class and Object in Java

- **Why**: Understanding classes and objects is fundamental to mastering object-oriented programming (OOP) in Java. They enable you to model real-world entities and their interactions in your code.
- **What**:
  - **Class**: A blueprint for creating objects. It defines a type, including fields and methods that define the behavior of objects.
  - **Object**: An instance of a class. It represents a real-world entity with state (attributes) and behavior (methods).
- **When**: Use classes and objects to model complex data structures and behaviors in a structured and modular way.
- **Example**:

```
public class Car {
    // Fields (attributes)
    String color;
    String model;
    int year;

    // Method (behavior)
    void startEngine() {
System.out.println("Engine started");
    }
```

```
}
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car();
myCar.color = "Red";
myCar.model = "Tesla Model S";
myCar.year = 2020;
myCar.startEngine();
    }
}
```

2. Constructors and Overloading

- **Why**: Constructors are special methods used to initialize objects. Overloading allows creating multiple constructors with different parameters, providing flexibility in object creation.
- **What**:
    - **Constructor**: A block of code that initializes a newly created object. It has the same name as the class and no return type.
    - **Overloading**: Defining multiple constructors with different para meter lists within the same class.
- **When**: Use constructors to ensure that objects are properly initialize d when created. Overload constructors to provide multiple ways to instantiate objects.

- **Example**:

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

    // Default constructor
    public Car() {
color = "Red";
        model = "Default Model";
        year = 2020;
    }

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

    void displayInfo() {
System.out.println("Color: " + color + ", Model: " + model + ", Year: " +
year);
    }
```

```
    public static void main(String[] args) {
        Car defaultCar = new Car();
        Car customCar = new Car("Blue", "Ford Mustang", 1967);
defaultCar.displayInfo();
customCar.displayInfo();
    }
}
```

3. Object Creation, Accessing Members of the Class

- **Why**: Creating objects and accessing their members is essential for utilizing the functionalities defined in a class.
- **What**:
  - **Object Creation**: Using the new keyword followed by the constructor.
  - **Accessing Members**: Using the dot . operator to access fields and methods of the object.
- **When**: Whenever you need to use the attributes and behaviors defined in a class.
- **Example**:

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

    void startEngine() {
```

```
System.out.println("Engine started");
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car();
        // Accessing fields
myCar.color = "Red";
myCar.model = "Tesla Model S";
myCar.year = 2020;
        // Accessing methods
myCar.startEngine();
System.out.println("Car: " + myCar.color + " " + myCar.model + " " +
myCar.year);
    }
}
```

4. this Keyword

- **Why**: The this keyword is used to refer to the current object instance . It's crucial for distinguishing between instance variables and parameters with the same name.
- **What**: this is a reference variable that refers to the current object.
- **When**: Use this when you need to refer to instance variables within

constructors, methods, or setters, especially when the parameter na mes conflict with instance variable names.

- **Example**:

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

    // Constructor
    public Car(String color, String model, int year) {
this.color = color; // Refers to the instance variable
this.model = model;
this.year = year;
    }
    void displayInfo() {
System.out.println("Color: " + color + ", Model: " + model + ", Year: " +
year);
    }
    public static void main(String[] args) {
        Car myCar = new Car("Blue", "Ford Mustang", 1967);
myCar.displayInfo();
    }
}
```

5. Methods in Java Theory

1. Defining Methods

- **Why**: Methods allow you to define reusable blocks of code that perform a specific task. They promote modularity and code reusability.
- **What**: A method is a block of code within a class that performs a specific task. It can be invoked or called to execute.
- **When**: Use methods to organize code into logical sections, perform repetitive tasks, or encapsulate functionality.
- **Syntax**:

```
returnTypemethodName(parameters) {
    // Method body
}
```

- **Example**:

```
public class Example {
    public void printMessage() {
System.out.println("Hello, World!");    // Method definition
    }
    public static void main(String[] args) {
        Example obj = new Example();
```

```
obj.printMessage(); // Method call
    }
}
```

2. Method Parameters and Return Types

- **Why**: Parameters allow methods to accept inputs, making them flexible and reusable. Return types enable methods to return results after execution.
- **What**:
    - **Parameters**: Variables listed in the method definition that accept values when the method is called.
    - **Return Type**: The data type of the value that the method return s. If no value is returned, use void.
- **When**: Use parameters to pass data to methods. Use return types to return processed data.
- **Syntax**:

```
returnTypemethodName(parameterTypeparameterName) {
    // Method body
    return value;
}
```

- **Example**:

```
public class Example {
    // Method with parameters and return type
    public int add(int a, int b) {
```

```
        return a + b;
    }
    public static void main(String[] args) {
        Example obj = new Example();

int sum = obj.add(5, 10); // Method call with arguments
System.out.println("Sum: " + sum); // Output: Sum: 15
    }
}
```

3. Method Overloading

- **Why**: Method overloading allows multiple methods with the same name but different parameters. It improves code readability and usability.
- **What**: Defining multiple methods with the same name but different parameter lists within the same class.
- **When**: Use method overloading when methods need to perform similar tasks but with different input parameters.
- **Syntax**:

```
public class Example {
    // Overloaded methods
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
```

```
        return a + b;
    }
    public static void main(String[] args) {
        Example obj = new Example();

System.out.println(obj.add(5, 10)); // Calls add(int, int)

System.out.println(obj.add(5.5, 10.5)); // Calls add(double, double)
    }
}
```

4. Static Methods and Variables

- **Why**: Static methods and variables belong to the class rather than any instance. They can be accessed directly using the class name, making them suitable for utility or helper methods and shared data.
- **What**:
  - **Static Method**: A method declared with the static keyword. It can be called without creating an instance of the class.
  - **Static Variable**: A variable declared with the static keyword. It is shared among all instances of the class.
- **When**: Use static methods for utility functions that do not depend on instance variables. Use static variables for constants or shared data across all instances.
- **Syntax**:

```
public class Example {
```

```
    // Static variable
    static int count = 0;

    // Static method
    public static void displayCount() {

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



}
    public static void main(String[] args) {
        // Accessing static method and variable
Example.count = 5;
Example.displayCount(); // Output: Count: 5
    }
}
```

6.Object-Oriented Programming (OOPs) Concepts Theory

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

- **Encapsulation**:
    - **Why**: Encapsulation is used to protect the internal state of an object and only expose what is necessary. This helps in reducing complexity and increasing reusability and maintainability.

- **What**: Encapsulation involves bundling the data (attributes) and methods (functions) that manipulate the data into a single unit, called a class. It restricts direct access to some of the object's components, which can be achieved through access modifiers.
- **When**: Use encapsulation to hide the internal state and require all interaction to be performed through an object's methods.
- **Example**:

```
public class EncapsulationExample {
    private String name;


   public String getName() {
       return name;
    }

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

- **Inheritance**:
  - **Why**: Inheritance is used to promote code reusability. It allows a new class to inherit the properties and behaviors of an existin g class.
  - **What**: Inheritance is a mechanism where one class (subclass) inherits the attributes and methods of another class(superclass)

- **When**: Use inheritance when there is a natural hierarchical rela tionship, and you want to reuse code from existing classes.
- **Example**:

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


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

- **Polymorphism**:
  - **Why**: Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. It simplifies code and improves reusability.
  - **What**: Polymorphism is the ability of a single function, operator, or object to operate in different ways based on context.
  - **When**: Use polymorphism to perform a single action in different ways.
  - **Example**:

```
public class Animal {
```

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

System.out.println("Bark");
    }
}
public class Cat extends Animal {
    @Override
    public void makeSound() {
System.out.println("Meow");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();
myDog.makeSound(); // Outputs: Bark
myCat.makeSound(); // Outputs: Meow
    }
```

```
}
```

- **Abstraction**:
  - **Why**: Abstraction is used to hide complex implementation details and only expose essential features. This helps in reducing complexity and focusing on high-level logic.
  - **What**: Abstraction is the process of hiding the internal details and showing only the functionality.
  - **When**: Use abstraction to separate implementation from the interface.
  - **Example**:

```
abstract class Animal {
    public abstract void makeSound();
}
public class Dog extends Animal {
    public void makeSound() {
System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
myDog.makeSound(); // Outputs: Bark
    }
}
```

2. Inheritance: Single, Multilevel, Hierarchical

- **Single Inheritance**:
  - **What**: A class inherits from one superclass.
  - **Example**:

```
public class Animal {
    // Superclass code
}

public class Dog extends Animal {
    // Subclass code
}
```

- **Multilevel Inheritance**:
  - **What**: A class is derived from another class, which is also derived from another class.
  - **Example**:

```
public class Animal {
    // Superclass code
}
public class Mammal extends Animal {
    // Subclass code
}
public class Dog extends Mammal {
    // Subclass code
}
```

- **Hierarchical Inheritance**:
    - **What**: Multiple classes inherit from a single superclass.
    - **Example**:

```
public class Animal {
   // Superclass code
}



public class Dog extends Animal {
   // Subclass code
}
public class Cat extends Animal {
   // Subclass code
}
```

3. Method Overriding and Dynamic Method Dispatch

- **Method Overriding**:
    - **Why**: Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass. This enables runtime polymorphism.
    - **What**: Overriding a method means defining a method in the subclass with the same name and parameters as in its superclass.
    - **When**: Use method overriding when you need to change or extend the behavior of an inherited method.

- **Example**:

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

```
public class Dog extends Animal {
    @Override
    public void makeSound() {
System.out.println("Bark");
    }
}
```

- **Dynamic Method Dispatch**:
  - **Why**: Dynamic method dispatch allows the JVM to determine at runtime which method to invoke. This is the cornerstone of runtime polymorphism in Java.
  - **What**: It is the process where a call to an overridden method is resolved at runtime rather than compile-time.
  - **When**: Use dynamic method dispatch when you want to implement polymorphic behavior.
  - **Example**:

```
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
myAnimal.makeSound(); // Outputs: Bark
    }
}
```

7.Constructors and Destructors Theory

1. Constructor Types (Default, Parameterized)

- **Why**: Constructors initialize objects when they are created, ensuring the object is in a valid state.
- **What**:
  - **Default Constructor**: A constructor that takes no arguments. It initializes the object with default values.
  - **Parameterized Constructor**: A constructor that takes argumen ts to initialize the object with specific values.
- **When**: Use default constructors for simple objects with default value s and parameterized constructors when you need to initialize objects with specific values.
- **Example**:

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

```
    // Default constructor
    public Car() {
color = "Red";
        model = "Default Model";
    }

    // Parameterized constructor
    public Car(String color, String model) {

this.color = color;
this.model = model;
    }

    public void displayInfo() {

System.out.println("Color: " + color + ", Model: " + model);
    }

    public static void main(String[] args) {
        Car defaultCar = new Car();
        Car customCar = new Car("Blue", "Ford Mustang");
defaultCar.displayInfo();
customCar.displayInfo();
```

```
    }
}
```

2. Copy Constructor (Emulated in Java)

- **Why**: A copy constructor creates a new object as a copy of an existing object. It is used to duplicate objects.
- **What**: Although Java doesn't have an explicit copy constructor, we can emulate it by writing a constructor that takes another object of the same class as an argument.
- **When**: Use copy constructors when you need to create a duplicate

  of an existing object with the same properties.
- **Example**:

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

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

    // Copy constructor
    public Car(Car car) {
```

```
this.color = car.color;
this.model = car.model;
    }

    public void displayInfo() {
System.out.println("Color: " + color + ", Model: " + model);
    }

    public static void main(String[] args) {

        Car originalCar = new Car("Blue", "Ford Mustang");
        Car copiedCar = new Car(originalCar);
originalCar.displayInfo();
copiedCar.displayInfo();
    }
}
```

3. Constructor Overloading

- **Why**: Constructor overloading allows a class to have more than one constructor with different parameter lists, providing multiple ways to initialize objects.
- **What**: Overloading means having multiple constructors with the same name but different parameter lists.
- **When**: Use constructor overloading to provide flexibility in object creation.

- **Example**:

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

    // Default constructor
    public Car() {
color = "Red";
        model = "Default Model";

        year = 2020;
    }

    // Parameterized constructor with two parameters
    public Car(String color, String model) {
this.color = color;
this.model = model;
this.year = 2020;
    }

    // Parameterized constructor with three parameters
    public Car(String color, String model, int year) {
this.color = color;
```

```
this.model = model;
this.year = year;
    }

    public void displayInfo() {
System.out.println("Color: " + color + ", Model: " + model + ", Year: " + year);
    }

    public static void main(String[] args) {
        Car defaultCar = new Car();

        Car modelCar = new Car("Blue", "Ford Mustang");
        Car customCar = new Car("Black", "Chevrolet Camaro", 1967);
defaultCar.displayInfo();
modelCar.displayInfo();
customCar.displayInfo();
    }
}
```

4. Object Life Cycle and Garbage Collection

- **Why**: Understanding the object life cycle and garbage collection is crucial for managing memory and ensuring that resources are allocated and deallocated properly.
- **What**:

- **Object Life Cycle**: The stages an object goes through from creation to garbage collection.
- **Garbage Collection**: The process by which Java reclaims memory by automatically removing objects that are no longer in use.

- **When**: The JVM manages the object life cycle, and garbage collection happens automatically. However, understanding this helps in writing efficient and memory-safe applications.
- **Example**:
  - **Object Creation**: An object is created using the new keyword.
  - **Object In Use**: The object is actively used in the program.
  - **Object Unreachable**: Once no references point to the object, it becomes eligible for garbage collection.
  - **Garbage Collection**: The JVM's garbage collector reclaims the memory.

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        Car car1 = new Car("Red", "Toyota"); // Object created
        car1 = null; // Object no longer referenced, eligible for garbage collection

        // Requesting JVM to run Garbage Collector
System.gc();
System.out.println("Garbage collection requested.");
```

```
    }

    @Override
    protected void finalize() throws Throwable {
System.out.println("Garbage collector called!");
System.out.println("Object garbage collected: " + this);
    }
}
```

8.Arrays and Strings Theory

1. One-Dimensional and Multidimensional Arrays

- **What**: Arrays are data structures that store multiple values of the sa me type in a single variable.
- **Why**: Arrays help manage collections of data efficiently, allowing for easy access and manipulation of elements.
- **When**: Use arrays when you need to store and work with multiple values of the same type.
- **One-Dimensional Array**:
    - **Syntax**:

```
int[] numbers = new int[5];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
numbers[4] = 5;
```

- **Example**:

```
public class OneDArrayExample {
    public static void main(String[] args) {
int[] numbers = {1, 2, 3, 4, 5};
        for (int i = 0; i<numbers.length; i++) {
System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}
```

- **Multidimensional Array**:
  - **Syntax**:

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

  - **Example**:

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

```
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.println("Element at [" + i + "][" + j + "]: " + matrix[i][j]);
            }
        }
    }
}
```

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

- **What**: Strings in Java are objects that represent sequences of characters. String, StringBuffer, and StringBuilder classes are used for string manipulation.
- **Why**: Efficient string handling and manipulation are essential for processing textual data.
- **When**: Use String for immutable strings, StringBuffer for mutable strings with thread safety, and StringBuilder for mutable strings without thread safety.

- **String Class**:
    - **Example**:

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

- **StringBuffer Class**:
    - **Example**:

```
public class StringBufferExample {
    public static void main(String[] args) {
StringBuffersb = new StringBuffer("Hello");
sb.append(", World!");

System.out.println(sb.toString());
    }
}
```

- **StringBuilder Class**:
    - **Example**:

```
public class StringBuilderExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
sb.append(", World!");
```

```
System.out.println(sb.toString());
    }
}
```

3. Array of Objects

- **What**: An array of objects is a collection where each element is an instance of a class.
- **Why**: Useful for managing and manipulating collections of objects.
- **When**: Use an array of objects when you need to work with multiple instances of a class.
- **Example**:

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



    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
    public void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
    public static void main(String[] args) {
        Car[] cars = new Car[3];
```

```
        cars[0] = new Car("Ford Mustang", 1967);
        cars[1] = new Car("Chevrolet Camaro", 1969);
        cars[2] = new Car("Dodge Charger", 1970);
            for (Car car : cars) {
        car.displayInfo();
            }
    }
}
```

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

- **What**: String methods are built-in functions that allow for various string manipulations.
- **Why**: They provide powerful tools for working with and transforming strings.

- **When**: Use these methods to perform specific operations on strings, such as finding length, accessing characters, and extracting substrings.
- **Common String Methods**:
  - **length()**: Returns the length of the string.

```
String str = "Hello";
int length = str.length(); // 5
```

  - **charAt()**: Returns the character at a specified index.

```
char ch = str.charAt(1); // 'e'
```

- **substring()**: Returns a new string that is a substring of the original string.

```
String substr = str.substring(1, 4); // "ell"
```

- **toUpperCase()**: Converts all characters in the string to uppercase.

```
String upperStr = str.toUpperCase(); // "HELLO"
```

- **toLowerCase()**: Converts all characters in the string to lowercase.

```
String lowerStr = str.toLowerCase(); // "hello"
```

- **contains()**: Checks if the string contains a specified sequence of characters.

```
boolean contains = str.contains("ell"); // true
```

- **replace()**: Replaces all occurrences of a specified character or sequence of characters with another.

```
String newStr = str.replace('l', 'p'); // "Heppo"
```

9. Inheritance and Polymorphism Theory

1. Inheritance Types and Benefits

- **What**: Inheritance is a mechanism in which one class acquires the properties (fields) and behaviors (methods) of another class.
- **Why**: Inheritance promotes code reusability and establishes a natural hierarchy between classes.

- **When**: Use inheritance when you need to create a new class that is a refined version of an existing class.
- **Types of Inheritance**:
    - **Single Inheritance**:
        - **What**: A subclass inherits from one superclass.
        - **Example**:

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

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

    - **Multilevel Inheritance**:
        - **What**: A class is derived from another class, which is also derived from another class.
        - **Example**:

```
public class Animal {
    public void eat() {
```

```
System.out.println("This animal eats food.");
    }
}
public class Mammal extends Animal {
    public void breathe() {
System.out.println("This mammal breathes air.");
    }
}
public class Dog extends Mammal {
    public void bark() {
System.out.println("The dog barks.");
    }
}
```

- **Hierarchical Inheritance**:
  - **What**: Multiple classes inherit from a single superclass.
  - **Example**:

```
public class Animal {
    public void eat() {

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

public class Dog extends Animal {
```

```
    public void bark() {
System.out.println("The dog barks.");
    }
}

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

- **Benefits of Inheritance**:
    - **Code Reusability**: Inheritance allows for the reuse of code, reducing redundancy and improving maintainability.
    - **Method Overriding**: Provides the ability to override methods to implement specific behavior in the subclass.
    - **Polymorphism**: Enables polymorphism, allowing objects to be treated as instances of their parent class.

2. Method Overriding

- **What**: Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.
- **Why**: It enables runtime polymorphism and allows the subclass to define specific behaviors.

- **When**: Use method overriding when you need to change or extend the behavior of an inherited method.
- **Example**:

```
public class Animal {
    public void makeSound() {
System.out.println("Animal sound");
    }
}
public class Dog extends Animal {
    @Override
    public void makeSound() {
System.out.println("Bark");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
myDog.makeSound(); // Outputs: Bark
    }
}
```

3. Dynamic Binding (Run-Time Polymorphism)

- **What**: Dynamic binding refers to the process where the method call is resolved at runtime rather than compile-time.

- **Why**: It allows for dynamic method dispatch, enabling the JVM to determine at runtime which method to invoke. This is fundamental to achieving runtime polymorphism.
- **When**: Use dynamic binding when you want to implement polymorphic behavior and have different implementations of a method in subclasses.
- **Example**:

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

public class Dog extends Animal {
    @Override
    public void makeSound() {
System.out.println("Bark");
    }
}



public class Cat extends Animal {
    @Override
```

```
    public void makeSound() {
System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;

myAnimal = new Dog();
myAnimal.makeSound(); // Outputs: Bark

myAnimal = new Cat();
myAnimal.makeSound(); // Outputs: Meow
    }
}
```

4. super Keyword and Method Hiding

- super **Keyword**:
    - **What**: The super keyword is used to access members (fields and methods) of the superclass from the subclass.
    - **Why**: It helps in invoking the superclass constructor, accessing superclass methods, and differentiating between superclass and subclass members.

- **When**: Use super to call superclass constructors, override methods, and access superclass fields when they are hidden by subclass fields.
- **Example**:

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

public class Dog extends Animal {
    public void eat() {
    super.eat();
    System.out.println("Dog eats");
  }
}
```

- **Method Hiding**:
  - **What**: Method hiding occurs when a subclass defines a static method with the same name and signature as a static method in the superclass. The superclass's static method is hidden in the subclass.
  - **Why**: It allows a subclass to define its version of a static method, effectively hiding the superclass's version.

- **When**: Use method hiding when you need to redefine a static method in a subclass.
- **Example**:

```
public class Animal {
    public static void makeSound() {
System.out.println("Animal sound");
    }
}

public class Dog extends Animal {
    public static void makeSound() {
System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
Animal.makeSound(); // Outputs: Animal sound
Dog.makeSound();    // Outputs: Bark
    }
}
```

10. Interfaces and Abstract Classes Theory

1. Abstract Classes and Methods

- **What**: An abstract class is a class that cannot be instantiated and is used to define a common interface for its subclasses. Abstract methods are declared without an implementation.
- **Why**: Abstract classes are useful for defining a common interface for related classes while allowing subclasses to provide specific implementations.
- **When**: Use abstract classes when you want to create a base class that provides some implementation but leaves other parts to be implemented by subclasses.
- **Example**:

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
    public void eat() {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal {
    @Override
    void makeSound() {
System.out.println("Bark");
    }
```

```
}

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

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();
myDog.makeSound(); // Outputs: Bark
myCat.makeSound(); // Outputs: Meow
myDog.eat(); // Outputs: This animal eats food.
    }
}
```

2. Interfaces: Multiple Inheritance in Java

- **What**: An interface is a reference type in Java, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors. They are used to achieve multiple inheritance in Java.

- **Why**: Java does not support multiple inheritance through classes to avoid complexity and ambiguity. Interfaces provide a way to implement multiple inheritance by allowing a class to implement multiple interfaces.
- **When**: Use interfaces when you want to define a contract that multiple classes can implement, ensuring they provide specific behavior.
- **Example**:

```
interface Animal {
    void makeSound(); // Interface method
}

interface Pet {
    void play(); // Interface method
}

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

    @Override
```

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


}
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
myDog.makeSound(); // Outputs: Bark
myDog.play(); // Outputs: Dog is playing.
    }
}
```

3. Implementing Multiple Interfaces

- **What**: Implementing multiple interfaces means that a class provides implementations for methods from more than one interface. This allows the class to inherit behavior from multiple sources.
- **Why**: It promotes flexibility and modularity, allowing a class to combine behaviors from multiple interfaces.
- **When**: Use multiple interfaces when a class needs to exhibit behaviors defined in different interfaces.
- **Example**:

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

```
interface Pet {
    void play();
}

class Cat implements Animal, Pet {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }

    @Override
    public void play() {
        System.out.println("Cat is playing.");
    }
}

public class Main {
    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.makeSound(); // Outputs: Meow
        myCat.play(); // Outputs: Cat is playing.
    }
}
```

12. Exception Handling Theory

1. Types of Exceptions: Checked and Unchecked

- **What**: Exceptions are events that disrupt the normal flow of a program. They can be categorized into checked and unchecked exceptions.
- **Why**: Understanding the types of exceptions helps in writing robust and error-free code by handling potential errors gracefully.
- **When**: Use checked exceptions for recoverable conditions and unchecked exceptions for programming errors.
- **Checked Exceptions**: These are exceptions that are checked at compile-time. The programmer is forced to handle these exceptions.
    - **Example**:

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

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

```
            File file = new File("test.txt");
            FileReaderfr = new FileReader(file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Unchecked Exceptions**: These are exceptions that are not checked at compile-time. They occur during runtime and are usually caused by programming errors.

    - **Example**:

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        // This will throw ArrayIndexOutOfBoundsException
        System.out.println(numbers[5]);
    }
}
```

2. try, catch, finally, throw, throws

- **What**: These keywords are used to handle exceptions in Java.
- **Why**: They provide a mechanism to handle runtime errors, ensuring the program can continue executing or terminate gracefully.
- **When**: Use these keywords to handle exceptions and maintain the

normal flow of the application.

- **try**: The block of code to be tested for exceptions.
    - **Syntax**:

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

- **catch**: The block of code to handle the exception.
    - **Syntax**:

```
catch (ExceptionType e) {
   // Code to handle the exception
}
```

- **finally**: The block of code that will always execute, regardless of whether an exception is thrown or not.
    - **Syntax**:

```
finally {
   // Code to be executed after try and catch blocks
}
```

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

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

- **throws**: Used in method signatures to declare that a method can throw exceptions.
    - **Syntax**:

```
public void methodName() throws ExceptionType {
    // Method code
}
```

- **Example**:

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


         try {
                int result = divide(10, 0);
                System.out.println("Result: " + result);
            } catch (ArithmeticException e) {
                System.out.println("Cannot divide by zero");


}
 finally {
        System.out.println("Execution completed");
        }
    }
    public static int divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }
```

}

3. Custom Exception Classes

- **What**: Custom exceptions are user-defined exceptions that extend the Exception class or its subclasses.
- **Why**: They provide a way to create meaningful and specific error me ssages and handling mechanisms for application-specific issues.

- **When**: Use custom exceptions when you need to handle specific err or conditions that are not covered by Java's built-in exceptions.
- **Example**:

```
// Custom exception class
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);

        }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
                validateAge(15);
        }
```

```
catch (CustomException e) {
        System.out.println("Caught custom exception: " + e.getMessage());
        }
    }

    public static void validateAge(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Age must be 18 or older");
        }
    }
}
```

13. Multithreading Theory

1. Introduction to Threads

- **What**: A thread is the smallest unit of a process that can be schedule d and executed by the CPU.
- **Why**: Threads allow a program to perform multiple tasks concurrently, improving performance and responsiveness.
- **When**: Use threads when you have tasks that can run independently and simultaneously.
- **Example**:

```
public class Main {
    public static void main(String[] args) {
System.out.println("Main thread is running...");
    }
```

```
}
```

2. Creating Threads by Extending Thread Class or Implementing Ru nnable Interface

- **Extending** Thread **Class**:
  - **What**: Creating a new class that extends the Thread class and overrides its run method.
  - **Why**: Simplifies the creation of a thread, but offers less flexibility compared to the Runnable interface.
  - **When**: Use this method when you want to directly create a new thread by extending the Thread class.
  - **Example**:

```
class MyThread extends Thread {
        public void run() {
            System.out.println("Thread is running...");
          }
}
public class Main {
   public static void main(String[] args) {
MyThread t1 = new MyThread();
        t1.start(); // Start the thread
   }
}
```

- **Implementing** Runnable **Interface**:
  - **What**: Implementing the Runnable interface and defining the

run method.

- **Why**: Offers more flexibility and is preferred when a class needs to extend another class.
- **When**: Use this method when you want to create a thread without extending the Thread class.
- **Example**:

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

public class Main {
    public static void main(String[] args) {
MyRunnablemyRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);

        t1.start(); // Start the thread
    }
}
```

3. Thread Life Cycle

- **What**: The stages a thread goes through during its lifetime.
- **Why**: Understanding the thread life cycle helps in managing thread states and synchronization.

- **When**: Use this knowledge to control thread behavior and optimize performance.
- **States in the Thread Life Cycle**:
  - **New**: A thread that has been created but not yet started.
  - **Runnable**: A thread that is ready to run or running.
  - **Blocked**: A thread that is blocked and waiting for a monitor lock
  - **Waiting**: A thread that is waiting indefinitely for another thread to perform a particular action.
  - **Timed Waiting**: A thread that is waiting for another thread to perform an action for up to a specified waiting time.
  - **Terminated**: A thread that has exited.

  - **Example**:

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

   public static void main(String[] args) {
MyThread t1 = new MyThread();
System.out.println("Thread state: " + t1.getState()); // New
       t1.start();
System.out.println("Thread state: " + t1.getState()); // Runnable
   }
```

```
}
```

4. Synchronization and Inter-thread Communication

- **What**: Synchronization ensures that only one thread accesses a resource at a time. Inter-thread communication allows threads to communicate and coordinate actions.
- **Why**: These mechanisms prevent data corruption and ensure consis tent results in multithreaded environments.
- **When**: Use synchronization to protect shared resources and inter-thread communication to enable cooperation between threads.
- **Synchronization**:
    - **Example**:

```
class Counter {

private int count = 0;
public synchronized void increment() {
        count++;
    }
public int getCount() {
        return count;
    }
}

class MyRunnable implements Runnable {
    private Counter counter;
```

```
    public MyRunnable(Counter counter) {
this.counter = counter;
    }

    public void run() {
        for (int i = 0; i< 1000; i++) {
counter.increment();
        }

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

        Counter counter = new Counter();
        Thread t1 = new Thread(new MyRunnable(counter));
        Thread t2 = new Thread(new MyRunnable(counter));
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final count: " + counter.getCount());
    }
}
```

- **Inter-thread Communication**:

- **Example**:

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

    public synchronized void produce(int value) throws
InterruptedException {
        while (available) {
wait();
        }
this.value = value;

available = true;
notify();
    }

    public synchronized int consume() throws InterruptedException {
        while (!available) {
                wait();
        }
        available = false;
notify();
        return value;
    }
```

```
}

class Producer implements Runnable {
    private SharedResourceresource;

    public Producer(SharedResource resource) {
        this.resource = resource;
    }

   public void run() {
        for (int i = 0; i< 10; i++) {

        try {
                resource.produce(i);
                System.out.println("Produced: " + i);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

class Consumer implements Runnable {
    private SharedResourceresource;
```

```
    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        for (int i = 0; i< 10; i++) {
            try {
                int value = resource.consume();
                System.out.println("Consumed: " + value);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread producerThread = new Thread(new Producer(resource));
        Thread consumerThread = new Thread(new Consumer(resource));

        producerThread.start();
        consumerThread.start();
```

```
    }
}
```

14. File Handling Theory

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

- **What**: File I/O (Input/Output) in Java is a mechanism to read from and write to files. It is part of the java.io package.
- **Why**: File I/O is essential for applications to interact with the filesystem, such as reading configurations, storing data, or logging in formation.
- **When**: Use file I/O whenever your application needs to persist data or read data from files.
- **Example**:

```
import java.io.File;

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

        File file = new File("example.txt");
        if (file.exists()) {
System.out.println("File exists");
        } else {
System.out.println("File does not exist");
        }
```

```
    }
}
```

2. FileReader and FileWriter Classes

- **What**: FileReader and FileWriter are classes used to read from and write to text files, respectively.
- **Why**: These classes provide simple methods to handle character-based file operations.
- **When**: Use FileReader for reading text files and FileWriter for writing text files.
- **FileReader Example**:

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

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

```
}
```

- **FileWriter Example**:

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

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

3. BufferedReader and BufferedWriter

- **What**: BufferedReader and BufferedWriter provide buffering for char acter input and output streams, improving performance.
- **Why**: Buffering reduces the number of I/O operations by storing data in memory before writing or after reading.
- **When**: Use these classes when you need efficient reading and writing of characters, arrays, and lines.
- **BufferedReader Example**:

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

```
import java.io.IOException;


public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReaderbr = new BufferedReader(new
FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
System.out.println(line);
            }
        } catch (IOException e) {
e.printStackTrace();
        }
    }
}
```

- **BufferedWriter Example**:

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


public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedWriterbw = new BufferedWriter(new
FileWriter("example.txt"))) {
```

```
bw.write("Hello, World!");
        } catch (IOException e) {
e.printStackTrace();
        }
    }
}
```

4. Serialization and Deserialization

- **What**: Serialization is the process of converting an object into a byte stream, allowing it to be easily stored or transmitted. Deserialization is the reverse process, converting a byte stream back into an object.
- **Why**: Serialization is used for persisting objects, sending objects over a network, and deep copying objects.
- **When**: Use serialization when you need to save the state of an object or transfer it between different parts of a program or different programs.
- **Example**:

```
import java.io.*;
class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;

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

```
this.age = age;
    }

    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        try {
            // Serialization
            Person person = new Person("Alice", 30);
FileOutputStreamfileOut = new FileOutputStream("person.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
out.writeObject(person);
out.close();
fileOut.close();
System.out.println("Serialized data is saved in person.ser");

            // Deserialization
FileInputStreamfileIn = new FileInputStream("person.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
            Person deserializedPerson = (Person) in.readObject();
```

```
in.close();
fileIn.close();
System.out.println("Deserialized Person: " + deserializedPerson);
        } catch (IOException | ClassNotFoundException e) {
e.printStackTrace();
        }
    }
}
```

Collections Framework Theory

1. Introduction to Collections Framework

- **What**: The Collections Framework is a unified architecture for repres enting and manipulating collections in Java. It includes interfaces, implementations, and algorithms to work with collections of objects.
- **Why**: It provides a standard way to handle groups of data, making it easier to store, manipulate, and retrieve collections efficiently.
- **When**: Use the Collections Framework when you need to manage groups of related objects, such as lists, sets, and maps.
- **Example**:

```
import java.util.ArrayList;
public class CollectionsExample {
```

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

2. List, Set, Map, and Queue Interfaces

- **List Interface**:
  - **What**: An ordered collection (also known as a sequence). Lists can contain duplicate elements.
  - **When**: Use when you need an ordered collection with access by index.
  - **Example**:

```
import java.util.List;
import java.util.ArrayList;
public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");
System.out.println(list);
```

```
    }
}
```

- **Set Interface**:
    - **What**: A collection that does not allow duplicate elements.
    - **When**: Use when you need a collection with unique elements.
    - **Example**:

```
import java.util.Set;
import java.util.HashSet;
public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
set.add("Apple"); // Duplicate, won't be added
System.out.println(set);
    }
}
```

- **Map Interface**:
    - **What**: A collection of key-value pairs. Keys are unique, and each key maps to exactly on e value.
    - **When**: Use when you need to associate keys with values.
    - **Example**:

```
import java.util.Map;
import java.util.HashMap;
public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
System.out.println(map);
    }
}
```

- **Queue Interface**:
    - **What**: A collection designed for holding elements prior to processing. Typically, it orders elements in a FIFO (First-In-First-Out) manner.
    - **When**: Use when you need to process elements in the order they were added.
    - **Example**:

```
import java.util.Queue;
import java.util.LinkedList;
public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
```

```
queue.add("Apple");
queue.add("Banana");
queue.add("Cherry");
System.out.println(queue.poll()); // Apple
System.out.println(queue);
    }
}
```

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

- **ArrayList**:
  - **What**: A resizable array implementation of the List interface.
  - **When**: Use when you need a dynamic array with fast random a ccess and iteration.
  - **Example**:

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

- **LinkedList**:

- **What**: A doubly linked list implementation of the List and Deque interfaces.
- **When**: Use when you need fast insertion and deletion from the beginning or end of the list.
- **Example**:

```
import java.util.LinkedList;
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");
System.out.println(list);
    }
}
```

- **HashSet**:
  - **What**: A hash table-based implementation of the Set interface.
  - **When**: Use when you need a collection with unique elements and fast access.
  - **Example**:

```
import java.util.HashSet;
public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
```

```
set.add("Apple");
set.add("Banana");
set.add("Cherry");
set.add("Apple"); // Duplicate, won't be added
System.out.println(set);
    }
}
```

- **TreeSet**:
    - **What**: A tree-based implementation of the Set interface that sorts elements in natural order.
    - **When**: Use when you need a sorted set with unique elements.
    - **Example**:

```
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String[] args) {
TreeSet<String> set = new TreeSet<>();
set.add("Banana");
set.add("Apple");
set.add("Cherry");
System.out.println(set); // Sorted set
    }
}
```

- **HashMap**:

- **What**: A hash table-based implementation of the Map interface.
- **When**: Use when you need a collection of key-value pairs with fast access.
- **Example**:

```
import java.util.HashMap;
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
System.out.println(map);
    }
}
```

- **TreeMap**:
  - **What**: A tree-based implementation of the Map interface that sorts keys in natural order.
  - **When**: Use when you need a sorted map of key-value pairs.
  - **Example**:

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

```
map.put("Banana", 2);
map.put("Apple", 1);
map.put("Cherry", 3);
System.out.println(map); // Sorted map
    }
}
```

4. Iterators and ListIterators

- **What**: Iterators are objects that allow you to traverse through a colle ction, one element at a time. ListIterator is a special type of iterator for lists that allows bidirectional traversal.
- **Why**: Iterators provide a standard way to loop through collections, making it easier to access and manipulate elements.
- **When**: Use iterators when you need to traverse a collection, especially when you need to remove elements during iteration.
- **Iterator**:
    - **Example**:

```
ArrayList<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
System.out.println(iterator.next());
}
```

- **ListIterator**:

- **Example**:

```
LinkedList<String> list = new LinkedList<>();
list.add("Apple");
list.add("Banana");
ListIterator<String>listIterator = list.listIterator();
while (listIterator.hasNext()) {
System.out.println(listIterator.next());
}
while (listIterator.hasPrevious()) {
System.out.println(listIterator.previous());
}
```

Java Input/Output (I/O) Theory

1. Streams in Java (InputStream, OutputStream)

- **What**: Streams in Java are sequences of data. InputStream is used to read data from a source, while OutputStream is used to write data to a destination.
- **Why**: Streams provide a standardized way to perform input and output operations, making it easy to read from and write to various data sources like files, network connections, etc.
- **When**: Use streams when you need to handle data flow between your program and external sources or destinations.
- **Example**:

```
import java.io.*;
public class StreamExample {
```

```
    public static void main(String[] args) {
        try {
            // Creating an InputStream to read data from a file
InputStreaminputStream = new FileInputStream("input.txt");

            // Creating an OutputStream to write data to a file
OutputStreamoutputStream = new FileOutputStream("output.txt");

            int data;
            while ((data = inputStream.read()) != -1) {
outputStream.write(data);
            }

inputStream.close();
outputStream.close();
        } catch (IOException e) {
e.printStackTrace();
        }
    }
}
```

2. Reading and Writing Data Using Streams

- **What**: Reading and writing data using streams involves handling bytes of data through InputStreams and OutputStreams.
- **Why**: Streams enable efficient handling of data input and output

operations, allowing for reading from and writing to files, network sockets, and other data sources.

- **When**: Use streams for low-level I/O operations where you need fine-grained control over data handling.
- **Example**:

```
import java.io.*;
public class DataStreamExample {
    public static void main(String[] args) {
        try {
            // Writing data to a file using FileOutputStream
FileOutputStreamfileOutputStream = new FileOutputStream("data.txt");
DataOutputStreamdataOutputStream = new
DataOutputStream(fileOutputStream);

dataOutputStream.writeInt(42);
dataOutputStream.writeDouble(3.14);
dataOutputStream.writeUTF("Hello, World!");

dataOutputStream.close();

            // Reading data from a file using FileInputStream
FileInputStreamfileInputStream = new FileInputStream("data.txt");
DataInputStreamdataInputStream = new
DataInputStream(fileInputStream);
```

```
            int intValue = dataInputStream.readInt();
            double doubleValue = dataInputStream.readDouble();
            String stringValue = dataInputStream.readUTF();

dataInputStream.close();

System.out.println("Read values: " + intValue + ", " + doubleValue + ", " +
stringValue);
        } catch (IOException e) {
e.printStackTrace();
        }
    }
}
```

3. Handling File I/O Operations

- **What**: File I/O operations involve reading from and writing to files using classes from the java.io package.
- **Why**: File I/O operations are essential for persistent data storage, configuration management, and handling user files.
- **When**: Use file I/O operations to store and retrieve data from files efficiently.
- **Example**:

```
import java.io.*;
public class FileIOExample {
```

```
    public static void main(String[] args) {
        try {
            // Writing to a file
            FileWriter writer = new FileWriter("example.txt");
writer.write("Hello, World!");
writer.close();

            // Reading from a file
FileReader reader = new FileReader("example.txt");
BufferedReaderbufferedReader = new BufferedReader(reader);
            String line;
            while ((line = bufferedReader.readLine()) != null) {
System.out.println(line);
            }
bufferedReader.close();
        } catch (IOException e) {
e.printStackTrace();
        }
    }
}
```