POLYMORPHISM

Definition:

Polymorphism in Java refers to the ability of an object to take on multiple forms. It is a core concept of Object-Oriented Programming (OOP) and enables one interface to be used for a general class of actions, with the specific action being determined at runtime.

Polymorphism allows Java to be flexible and extensible, as the same method can behave differently depending on the object it is acting upon.

Types of Polymorphism in Java:

1. **Compile-Time Polymorphism**
2. **Run-Time polymorphism**

COMPILE-TIME POLYMORPHISM:

- **Achieved through method overloading or constructor overloading.**
- **In method overloading, multiple methods with the same name are defined, but with different parameter types or numbers.**

- The method to be called is determined at compile-time based on the method signature.

Example:

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

RUN-TIME POLYMORPHISM:

- **Achieved through method overriding.**
- **In method overriding, a subclass provides a specific implementation of a method that is already defined in its superclass.**

- The method to be called is determined at runtime based on the object's actual type, not the reference type.

Example:

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

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

class Cat extends Animal {
    @Override
```

```
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();  // Reference type is Animal, but object type is Dog
        animal.sound();  // Outputs "Dog barks"
    }
}
```

KEYPOINTS:

- **Method Overloading is used to achieve compile-time polymorphism.**
- **Method Overriding is used to achieve runtime polymorphism.**
- **Dynamic method dispatch happens during runtime to**

decide which method to invoke, based on the actual object.

BENEFITS OF POLYMORPHISM:

- **Flexibility: It allows you to write code that can work with objects of various types in a uniform way.**
- **Maintainability: Easier to modify and extend as new classes can be added without changing existing code.**

INHERITANCE

Definition:

Inheritance is one of the key features of Object-Oriented Programming (OOP) in Java. It allows a new class to inherit the properties (fields) and behaviors (methods) of an existing class, enabling code reusability and the creation of

Inheritance is one of the key features of Object-Oriented Programming (OOP) in Java.

It allows a new class to inherit the properties (fields) and behaviors (methods) of an existing class, enabling code reusability and the creation of hierarchical relationships

between classes

Key Concepts of Inheritance:

1. **Superclass (Parent Class)**: The class whose properties and methods are inherited by another class.
2. **Subclass (Child Class)**: The class that inherits properties and methods from another class.
3. **`extends` Keyword**: The keyword used to indicate inheritance in Java.

TYPES OF INHERITANCE:

1. **SINGLE INHERITANCE**
2. MULTIPLE INHERITANCE
3. MULTILEVEL INHERITANCE
4. HYBRID INHERITANCE
5. HIERARCHICAL INHERTANCE

SINGLE INHERITANCE:

 Single Inheritance: A class can inherit from one superclass.

- Java supports only single inheritance for classes (no multiple inheritance for classes).

 MULTIPLE INHERITANCE: A Single class has multiple base classes

Multilevel Inheritance: A class can inherit from another class, which in turn inherits from another class, forming a chain of inheritance.

Example: `class A -> class B -> class C`

Hierarchical Inheritance: Multiple subclasses can inherit from the same superclass.

Example: `class A -> class B, class C`

Hybrid Inheritance: A combination of the above types. In Java, this is typically achieved using interfaces.

INTERFACES

Definition:

In Java, interfaces are a powerful feature of Object-Oriented Programming (OOP) that allow you to define a contract for classes without providing the implementation. They provide a way to achieve abstraction and multiple inheritance in Java, since a class can implement multiple interfaces.

This allows Java to avoid the complexities of multiple inheritance while still providing a mechanism for polymorphism and code flexibility.

Key Characteristics of Interfaces in Java:

1. **Abstract Methods**: All methods in an interface are implicitly abstract (until Java 8).

2. **No Constructors**: An interface cannot have constructors because it cannot be instantiated.

3. **No Instance Variables**: You cannot define instance variables in an interface (only static and final variables).

4. **Multiple Implementations**: A class can implement multiple interfaces, which is a way to achieve **multiple inheritance**.

5. **Default Methods (Java 8)**: Since Java 8, interfaces can have **default methods** with a body. This allows you to provide default behavior for methods in the interface.

6. **Static Methods (Java 8)**: Interfaces can also have static methods.

7. **Private Methods (Java 9)**: Java 9 introduced **private methods** in interfaces to help with code reuse within default methods.

Syntax of an Interface in Java:

```
interface MyInterface{
    void myMethod();
    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
    static void staticMethod() {
```

```
        System.out.println("This is a static method.");
    }
}
```

Keypoints:

Interfaces define a contract, but do not provide implementation (until Java 8 with default methods).

A **class** implements an interface using the implements keyword.

Interfaces allow Java to achieve **multiple inheritance** through **multiple interface implementation**.

Default methods and **static methods** allow more functionality in interfaces.

Functional interfaces are commonly used in lambda expressions.

- **Private methods** help with code reuse in interfaces (Java 9+).

APPLET AND SWING

APPLET: An **applet** in Java is a small application designed to be embedded into a web page and run in the context of a web browser. Applets were once used for creating interactive features on websites, such as games, animations, or small tools. However, applets have largely been deprecated in favor of more modern web technologies like JavaScript, HTML5, and CSS. Major browsers have dropped support for applets due to security concerns, but understanding them can still be useful if you're learning Java or studying legacy systems.

Structure of a Basic Java Applet:

An applet is a subclass of the Applet class (or JApplet for Swing-based applets). Here's an example of a simple applet written in Java:

APPLET EXAMPLE:

```
import java.applet.Applet;
import java.awt.Graphics;
```

```
public class HelloWorldApplet extends Applet{
public void init() {
    }
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 50, 50);
    }
}
```

KEYPOINTS:

1. **They require a special HTML <applet> tag to be embedded in a web page.**
2. **The applet lifecycle includes init(), start(), stop(), and destroy(), although only init() and paint() are commonly used.**

SWING:

Swing is a part of Java's javax.swing package and provides a set of GUI (Graphical User Interface) components for building window-based applications in Java. Swing is a part of Java Foundation Classes (JFC) and is an improvement over AWT (Abstract Window Toolkit). It offers more sophisticated and flexible UI elements than AWT, including features like customizable

look-and-feel, lightweight components, and better event handling.

KEY FEATURES IN SWING:

1.Lightweight Components: Unlike AWT, which relies on the native operating system to render components (heavyweight), Swing components are rendered by the Java runtime, making them lightweight and more portable.

2. Customizable Look-and-Feel: Swing allows you to change the appearance of your GUI using different "look-and-feel" settings, such as the Metal Look-and-Feel, Windows, or a cross-platform look.

3. Event Handling: Swing provides a robust event-handling model to capture and respond to user input such as mouse clicks, keyboard presses, and window events.

4. MVC Architecture: Swing follows a Model-View-Controller (MVC) architecture, where the components like buttons, labels, etc., separate their data model, display, and user actions.

5. Rich GUI Components: Swing provides a variety of components, including buttons, labels, text fields, combo boxes, tables, trees, and more.

SWING APPLICATION:

1. Creating a JFrame: JFrame is the main container for creating a window.

2. Adding Components: Swing components are added to the JFrame container.

3. Event Handling: You add listeners to components to handle user input events.

OOPS IN JAVA

Definition:

Object-Oriented Programming (OOP) in Java is a programming paradigm based on the concept of "objects," which contain both data and methods. OOP in Java is one of the most important concepts because it

allows for the design of modular, reusable, and maintainable software. Java is inherently an object-oriented language, meaning it encourages the use of classes and objects.

Core Concepts of OOP in Java:

1. **Class**: A class is a blueprint or prototype from which objects are created. It defines the properties (fields) and behaviors (methods) that objects created from the class will have.

   EXAMPLE:

```
public class Car{
    String color;
    String model;
    int year;
    public void startEngine() {
        System.out.println("Engine started!");
    }

    public void stopEngine() {
        System.out.println("Engine stopped!");
```

```
    }
}
```

2. **Object: An object is an instance of a class. When a class is defined, no memory is allocated for it, but when you create an object (instance) of that class, memory is allocated.**

Example:

```
        public class Main {
    public static void main(String[] args)
{
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.model = "Toyota";
        myCar.year = 2021;
        myCar.startEngine();
    }
}
```

3. **Encapsulation: Encapsulation is the concept of wrapping the data (variables) and methods into a single unit called a class. It also means restricting access to some of the**

object's components, typically using access modifiers like private, protected, or public.

Private fields: These variables cannot be accessed directly outside the class.

Public methods: These methods are used to provide access to the private fiel

ds (via getters and setters).

EXAMPLE:

```
 public class Person {
private String name;
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
}
```

-

PERIYAR MANIAMMAI
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University)
Established Under Sec. 3 of UGC Act, 1956 • NAAC Accredited
think • innovate • transform

NAME :A.NIVETHA

REG NO :123011019021

BRANCH :B.Tech CSE(Spec inAI&ML)

SUBJECT :OBJECT ORIENTED PROGRAMMING LANGUAGE

COURSE CODE :XCS304

ASSIGNMENT:CASE STUDY ON OOPS