Program &Class : B.Tech (CSE) & E – Section **Sem & Year : I & I**

Course code & Course Name : ACSD01 - Object Oriented Programming

Faculty : Dr.S.Sathees Kumar (IARE10907)

ANSWERS FOR TUTORIAL QUESTION BANK

MODULE -IV

INHERITANCE AND POLYMORPHISM

PART A-PROBLEM SOLVING AND CRITICAL THINKING QUESTIONS

1. What are the limitations of inheritance?

Inheritance in object-oriented programming brings several advantages, but it also has limitations. Here are some common limitations of inheritance:

1. **Limited Reusability:**
   - Inheritance promotes code reuse by allowing a subclass to inherit properties and behaviours from a superclass. However, sometimes a subclass may inherit more than it needs, leading to a less modular and more tightly coupled design.
2. **Diamond (Multiple) Inheritance Issues:**
   - Some programming languages, like Java, don't support multiple inheritance for classes (where a class can inherit from more than one class). This is to avoid the "diamond problem" or ambiguity that arises when a class inherits from two classes that have a common ancestor.

```
// Example of diamond problem in Java
class A {
   void display() {
      System.out.println("A");
   }
}

class B extends A {
   void display() {
      System.out.println("B");
   }
}
```

```
class C extends A {
   void display() {
      System.out.println("C");
   }
}

class D extends B, C { // Error: Multiple inheritance not allowed
}
```

1. **Tight Coupling:**
   - Inheritance can lead to tight coupling between subclasses and superclasses. Changes in the superclass may impact the subclasses, making the code more difficult to maintain and modify.
2. **Increased Complexity:**
   - As the depth of the inheritance hierarchy increases, it can become challenging to understand and manage the relationships between classes. This can result in increased complexity and difficulty in debugging.
3. **Brittle Base Class Problem:**
   - Modifying the base class (superclass) can impact all the derived classes. If a change is made to the base class, it may require modifications to all the derived classes, which can be error-prone and time-consuming.
4. **Security Concerns:**
   - Inheritance can expose the internal details of a class, leading to potential security risks. Subclasses may access or override methods that were not intended to be modified.
5. **Performance Overhead:**
   - Dynamic method dispatch, which allows objects of different classes to be treated as objects of a common superclass, can introduce a slight performance overhead compared to static method dispatch.

It's essential to carefully consider these limitations and design class hierarchies thoughtfully to ensure a balance between code reuse and maintainability.

2. How is an abstract class different from an interface?

An abstract class and an interface are both constructs in object-oriented programming that provide a way to achieve abstraction, but they have some key differences:

Keyword:

- Declared using the **'abstract'** keyword.

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

1. **Methods:**
    - Can have abstract and non-abstract (concrete) methods.
    - May have instance variables (fields).
    - Can have constructors.
    - Can have method implementations.
2. **Constructor:**
    - Can have a constructor, and it gets invoked when an instance of the concrete subclass is created.
3. **Access Modifiers:**
    - Can have access modifiers for methods (public, private, protected, etc.).
    - Can have abstract and non-abstract methods with various access modifiers.
4. **Multiple Inheritance:**
    - Supports single inheritance only. A class can extend only one abstract class.

Interface:

1. **Keyword:**
    - Declared using the **'interface'** keyword.

```
interface Animal {
   // Abstract method (implicitly public and abstract)
   void makeSound();
}
```

1. **Methods:**
    - Can have only abstract methods (implicitly public and abstract).
    - Cannot have instance variables (fields) until Java 8, which introduced default and static methods.
2. **Constructor:**
    - Cannot have constructors. Interfaces do not participate in object instantiation.
3. **Access Modifiers:**
    - All methods are implicitly public and abstract. Fields (after Java 8) can have access modifiers.
4. **Multiple Inheritance:**
    - Supports multiple inheritance. A class can implement multiple interfaces.

When to Use Which:

- **Abstract Class:**
  - Use when there is a need for a common base class that provides a partial implementation.
  - When you want to share code among related classes.
  - When you need to declare fields (instance variables) in the base class.
- **Interface:**
  - Use when defining a contract that multiple unrelated classes should adhere to.
  - When you want to achieve multiple inheritances.
  - When you want to create a lightweight, multiple-method contract without any implementation.

3. Why do we need to use inheritance?

Inheritance is a crucial concept in object-oriented programming (OOP) that provides several benefits, making it an essential tool in software development. Here are some key reasons why inheritance is important:

1. **Code Reusability:**
   - Inheritance allows a new class (subclass or derived class) to inherit properties and behaviours from an existing class (superclass or base class). This promotes code reuse, reducing redundancy and making the code more modular.
2. **Extensibility:**
   - New classes can be created by extending existing classes. This facilitates the addition of new features or modifications to existing functionalities without affecting the original code. It supports the evolution and growth of software systems.
3. **Polymorphism:**
   - Inheritance enables polymorphism, allowing objects of different classes to be treated as objects of a common superclass. This flexibility simplifies code and promotes a more generic and adaptable design.
4. **Organizing Code:**
   - Inheritance provides a way to organize code hierarchically. Classes can be structured in a hierarchy, reflecting relationships and dependencies between different entities in the system. This hierarchical organization enhances the clarity and maintainability of code.

4. How will you prove that the features of superclass are inherited in subclass?

To prove that the features of a superclass are inherited in a subclass, you can create a simple example in an object-oriented programming language such as Java. In this example, I'll use Java to illustrate the concept of inheritance:

```
// Superclass (Base class)
class Animal {
    // Attribute
    String species;

    // Constructor
    public Animal(String species) {
        this.species = species;
    }

    // Method
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

// Subclass (Derived class)
class Dog extends Animal {
    // Additional attribute specific to Dog
    String breed;

    // Constructor
    public Dog(String species, String breed) {
        // Calling the superclass constructor
        super(species);
        this.breed = breed;
    }

    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }

    // Additional method specific to Dog
    public void wagTail() {
        System.out.println("Dog is wagging its tail");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        // Creating an instance of the subclass
        Dog myDog = new Dog("Canine", "Golden Retriever");

        // Accessing inherited attribute from the superclass
        System.out.println("Species: " + myDog.species);

        // Calling inherited method from the superclass
        myDog.makeSound();

        // Accessing subclass-specific attribute
        System.out.println("Breed: " + myDog.breed);

        // Calling subclass-specific method
        myDog.wagTail();
    }
}
```

In this example:

- The '**Dog'** class extends the **'Animal'** class, inheriting the **'species'** attribute and the **makeSound** method from the superclass.
- The '**Dog'** class adds a specific attribute **(breed)** and a specific method **(wagTail).**
- In the **'Main'** class, we create an instance of '**Dog'** and demonstrate how it inherits features from the superclass and has its own specific attributes and methods.

5. Will a constructor and instance initialization blocks inherited to subclass?

In Java, constructors and instance initialization blocks are not inherited by a subclass. However, when a subclass is created, it implicitly calls the constructor of its superclass as part of its own construction process. This is done using the '**super()'** keyword, which invokes the constructor of the immediate superclass.

If the superclass has a no-argument constructor, it will be called automatically unless you explicitly call another constructor using '**super(...)'**. If the superclass does not have a no-argument constructor, and you don't provide a call to a specific constructor using '**super(...)',** then the compiler will generate an error.

Initialization blocks, whether static or instance, are not inherited by subclasses. They are specific to the class in which they are declared. However, they are executed in the order in which they appear in the class hierarchy. Specifically, the static initialization blocks of the superclass are executed when the class is loaded, and the instance initialization blocks are executed when an instance of the class is created.

Here's a simple example to illustrate:

```
class Superclass {
static {
    System.out.println("Static Initialization Block in Superclass");
}
{
    System.out.println("Instance Initialization Block in Superclass");
}
public Superclass() {
    System.out.println("Constructor in Superclass");
}
}
class Subclass extends Superclass {
static {
    System.out.println("Static Initialization Block in Subclass");
}

 {
    System.out.println("Instance Initialization Block in Subclass");
}
public Subclass() {
    super(); // Implicit call to the superclass constructor
    System.out.println("Constructor in Subclass");
  }
 }

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

output

Static Initialization Block in Superclass
Instance Initialization Block in Superclass
Constructor in Superclass
Static Initialization Block in Subclass
Instance Initialization Block in Subclass
Constructor in Subclass

As you can see, the static and instance initialization blocks are executed in the order they appear in the class hierarchy, but they are not inherited by the subclass.

6. Are static members inherited to subclass?

Static members, including static fields and static methods, are inherited by subclasses in Java. However, they are not overridden in the same way as instance members. Instead, they are accessible through the subclass using the subclass name or an instance of the subclass.

Here's an example to illustrate this:

```
class Superclass {
    static int staticField = 10;

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

class Subclass extends Superclass {
    // Subclass inherits the staticField and staticMethod from Superclass
}

public class Main {
    public static void main(String[] args) {
        // Accessing static field through subclass
        System.out.println("Static field in Subclass: " + Subclass.staticField);

        // Accessing static method through subclass
        Subclass.staticMethod();
    }
}
```

In this example, the **'Subclass'** inherits the '**staticField'** and '**staticMethod'** from the **Superclass**. You can access them using the subclass name **('Subclass')** without creating an instance of the subclass.

output

Static field in Subclass: 10
Static method in Superclass

So, while static members are inherited, they are not overridden in the same way that instance members are. Each class has its own copy of static members, and they are accessed through the class name rather than through an instance of the class.

7. What is the order of calling constructors in case of inheritance?

In Java, when a subclass is instantiated, the constructors are called in a specific order due to the process of inheritance. The order of constructor calls is as follows:

1. **Superclass Constructor:**
   - The constructor of the immediate superclass is called first. If the superclass has a no-argument constructor, it will be called automatically. If the superclass does not have a no-argument constructor and you don't explicitly call another constructor using super(...), then the compiler will generate an error.
2. **Instance Initialization Blocks (Superclass):**
   - After the superclass constructor is executed, any instance initialization blocks in the superclass are executed in the order they appear.
3. **Subclass Constructor:**
   - Next, the constructor of the subclass is called. If the subclass constructor does not explicitly call a superclass constructor using '**super(...)',** the default no-argument constructor of the superclass is called automatically.
4. **Instance Initialization Blocks (Subclass):**
   - After the subclass constructor is executed, any instance initialization blocks in the subclass are executed in the order they appear.

Here's an example to illustrate this order:

```
class Superclass {
   static {
      System.out.println("Static Initialization Block in Superclass");
   }
   {
      System.out.println("Instance Initialization Block in Superclass");
   }
   public Superclass() {
      System.out.println("Constructor in Superclass");
   }
}
```

```
class Subclass extends Superclass {
   static {
      System.out.println("Static Initialization Block in Subclass");
   }
   {
      System.out.println("Instance Initialization Block in Subclass");
   }

   public Subclass() {
      super(); // Implicit call to the superclass constructor
      System.out.println("Constructor in Subclass");
   }
}
public class Main {
   public static void main(String[] args) {
      Subclass obj = new Subclass();
   }
}
```

output

```
Static Initialization Block in Superclass
Instance Initialization Block in Superclass
Constructor in Superclass
Static Initialization Block in Subclass
Instance Initialization Block in Subclass
Constructor in Subclass
```

This output demonstrates the order in which the static and instance initialization blocks, as well as the constructors, are called during the instantiation of a subclass.

8. How will you restrict a member of a class from inheriting its subclass?

In Java, you can use the **'final'** keyword to restrict a member of a class from being inherited by its subclasses. When you declare a method, field, or class as '**final'**, it means that it cannot be overridden, extended, or inherited by any subclass.

Here are examples for different cases:

Final Method

```
class Parent {
   final void finalMethod() {
      // Some implementation
   }
}

class Child extends Parent {
```

```
    // Error: Cannot override final method
    // void finalMethod() { }
}
```

Final Field

```
class Parent {
    final int finalField = 10;
}

class Child extends Parent {
    // Error: Cannot override final field
    // int finalField = 20;
}
```

Final Class (Entire Class)

```
final class FinalClass {
    // Some implementation
}

// Error: Cannot inherit from final class
// class ChildClass extends FinalClass { }
```

By marking a member or class as **'final'**, you explicitly indicate that it should not be altered or extended by subclasses. Keep in mind that you can use **'final'** for methods, fields, or entire classes, and it serves to enforce constraints on inheritance.

9. What happens if both superclass and subclass have a field with the same name?

If both the superclass and subclass have a field with the same name, this is known as field hiding. In Java, when you have a field with the same name in both the superclass and subclass, the field in the subclass hides the field in the superclass. This means that when you refer to the field using an instance of the subclass, the subclass's field is accessed.

Here's an example to illustrate this:

```
class Superclass {
    int commonField = 10;
}

class Subclass extends Superclass {
    int commonField = 20;
```

```
    void displayFields() {
        System.out.println("Superclass Field: " + super.commonField);
        System.out.println("Subclass Field: " + this.commonField);
    }
}

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

output

Superclass Field: 10
Subclass Field: 20

In this example, both '**Superclass'** and '**Subclass'** have a field named '**commonField'**. When the '**displayFields'** method is called on an instance of '**Subclass',** it prints the values of both fields.'**super.commonField'** refers to the field in the superclass, and '**this.commonField'** refers to the field in the subclass.

It's important to note that field hiding can sometimes lead to confusion, and it's generally recommended to avoid using the same field names in both the superclass and subclass unless there's a specific reason to do so. If you want to access the superclass's field from the subclass, you can use the '**super'** keyword to differentiate between the fields.

10. Can you call the base class method without creating an instance?

In Java, you cannot call an instance method of a base class without creating an instance of that class. However, you can call a static method of a base class without creating an instance, as static methods belong to the class itself rather than instances of the class.

Here's an example to illustrate the difference:

```
class BaseClass {
    void instanceMethod() {
        System.out.println("Instance method in BaseClass");
    }

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

```
public class Main {
   public static void main(String[] args) {
      // Creating an instance of BaseClass
      BaseClass instance = new BaseClass();

      // Calling instance method through an instance
      instance.instanceMethod();  // This is how you call an instance method

      // Calling static method without creating an instance
      BaseClass.staticMethod();  // This is how you call a static method
   }
}
```

In the example above, '**instanceMethod'** is an instance method, so you need to create an instance of the class to call that method. On the other hand, '**staticMethod'** is a static method, so you can call it using the class name without creating an instance.

PART-B LONG ANSWER QUESTIONS

1. Briefly explain types of inheritance used in object oriented Programming.

In object-oriented programming (OOP), inheritance is a mechanism that allows a class to inherit properties and behaviour s from another class. There are several types of inheritance:

1. Single Inheritance:

- A class can inherit from only one superclass. In other words, there is a one-to-one relationship between a subclass and its superclass. Java supports single inheritance.

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

2. Multiple Inheritance:

- A class can inherit from more than one superclass. While this is conceptually appealing, it can lead to the **"diamond problem"** where the same method or field is inherited from multiple paths. Many languages, including Java, do not support multiple inheritance to avoid this issue.

```
// This is not allowed in Java
// class A { /* ... */ }
// class B { /* ... */ }
// class C extends A, B { /* ... */ }
```

3. Multilevel Inheritance:

- A class can inherit from another class, and then another class can inherit from it. This creates a hierarchy of classes.

```
class A { /* ... */ }
class B extends A { /* ... */ }
class C extends B { /* ... */ }
```

4. Hierarchical Inheritance:

- Multiple classes inherit from a single superclass. It represents a hierarchy where a common base class is extended by multiple subclasses.

```
class A { /* ... */ }
class B extends A { /* ... */ }
class C extends A { /* ... */ }
```

5. Hybrid (or Multiple) Inheritance:

- A combination of two or more types of inheritance. For example, a combination of single and multiple inheritance. While Java doesn't support multiple inheritance for classes, it allows a class to implement multiple interfaces, achieving a form of multiple inheritance through interfaces.

```
interface A { /* ... */ }
interface B { /* ... */ }
class C implements A, B { /* ... */ }
```

Inheritance is a fundamental concept in OOP that promotes code reuse, modularity, and extensibility. It allows you to create a new class by building upon an existing class, incorporating its features and potentially adding or modifying functionality.

2. Differentiate between multiple inheritance and multilevel inheritance.

Multiple Inheritance and Multilevel Inheritance are two different concepts in the realm of object-oriented programming:

1. Multiple Inheritance:

- **Definition:** Multiple Inheritance refers to the ability of a class to inherit from more than one superclass. That is, a class can have multiple parent classes.
- **Example (in a language supporting multiple inheritance):**

```
class A { /* ... */ }
class B { /* ... */ }
class C extends A, B { /* ... */ }  // This syntax is not allowed in Java
```

Issues: Multiple Inheritance can lead to the "diamond problem," where a class inherits from two classes that have a common ancestor. This can result in ambiguity when trying to access shared features from the common ancestor.

Multilevel Inheritance:

- Definition: Multilevel Inheritance refers to the chaining of inheritance in a hierarchy, where a class serves as the superclass for another class, and that second class becomes the superclass for a third class, and so on.
- **Example:**
  class A { /* ... */ }
  class B extends A { /* ... */ }
  class C extends B { /* ... */ }

Explanation: In this example, class **'B'** extends class **'A'**, and class **'C'** extends class **'B'**. This creates a multilevel hierarchy where each class builds upon the features of its immediate superclass.

In summary, the key difference is in the number of classes involved in the inheritance relationship:

- **Multiple Inheritance:** A class inherits from more than one superclass.
- **Multilevel Inheritance:** Classes are arranged in a hierarchy, and each class in the hierarchy serves as the superclass for the class below it.

3. What is the difference between inheritance and encapsulation?

Inheritance and encapsulation are two fundamental concepts in object-oriented programming (OOP), but they serve different purposes and focus on different aspects of designing and organizing code.

Inheritance:

- Definition: Inheritance is a mechanism in OOP that allows a new class (subclass or derived class) to inherit properties and behaviour s (fields and methods) from an existing class (superclass or base class).
- Purpose: It promotes code reuse, extensibility, and the creation of a hierarchy of classes. Subclasses can reuse the code of their superclasses and can also add or override methods to modify or extend behaviour .
- **Example:**

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

Key Concept: "Is-a" relationship. For example, a **Dog** "is a" kind of **Animal**.

Encapsulation:

- **Definition:** Encapsulation is the bundling of data (attributes or fields) and methods that operate on that data into a single unit called a class. It involves restricting access to the internal state of an object and only allowing interactions through well-defined public interfaces (methods).
- **Purpose:** It helps in data hiding, abstraction, and maintaining the integrity of the object by controlling access to its internal implementation details.
- **Example:**

```
public class Circle {
    private double radius;

    public void setRadius(double radius) {
        if (radius > 0) {
            this.radius = radius;
        } else {
            System.out.println("Invalid radius");
        }
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Key Concept: "Encapsulation hides the implementation details of an object."

In summary:

- **Inheritance:** Focuses on the relationship between classes, promoting code reuse and hierarchy.
- **Encapsulation:** Focuses on bundling data and methods into a single unit, providing a way to control access to the internal state of an object.

While they serve different purposes, they are often used together to create well-organized and maintainable object-oriented code.

4. Discuss the difference between inheritance and abstraction with real time examples.

- **Definition:** Inheritance is a mechanism in object-oriented programming that allows a new class (subclass) to inherit properties and behaviour s from an existing class (superclass). The subclass can reuse the code of the superclass and can also add or override methods to modify or extend behaviour .
- **Real-Time Example:** Consider a scenario where you have a base class called **Vehicle** with common properties and methods related to vehicles. You can then have subclasses like '**Car'** and **'Motorcycle'** that inherit from the Vehicle class. Each

subclass can have its specific features while inheriting the common properties and behaviour s from the **'Vehicle'** superclass.

```
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

class Motorcycle extends Vehicle {
    void ride() {
        System.out.println("Motorcycle is riding");
    }
}
```

Here, '**Car'** and '**Motorcycle'** inherit the **'start'** method from the '**Vehicle'** class, and they can have their own specific methods ('**drive'** and '**ride'**, respectively).

Abstraction:

- **Definition:** Abstraction is the process of hiding the complex implementation details of an object and exposing only the essential features. It involves creating abstract classes or interfaces that define a common structure without specifying the details of how it should be implemented.
- **Real-Time Example:** Consider a scenario where you want to model different shapes, such as circles, rectangles, and triangles. You can create an abstract class or interface called **'Shape'** that declares common methods like **'calculateArea'** and **'draw'**. The actual implementation details of these methods will be provided by the concrete subclasses.

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

class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
```

```
    }

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    double calculateArea() {
        return length * width;
    }

    @Override
    void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

- Here, the **'Shape'** class defines the common structure for all shapes, and the concrete subclasses (**'Circle'** and **'Rectangle'**) provide their own implementations for the **'calculateArea'** and **'draw'** methods.

In summary:

- **Inheritance** focuses on the relationship between classes and the reuse of code.
- **Abstraction** focuses on hiding implementation details and providing a common interface or structure.

5. What is the difference between polymorphism and inheritance?

Polymorphism and inheritance are two key concepts in object-oriented programming, but they serve different purposes:

1. Inheritance:

- **Definition:** Inheritance is a mechanism that allows a class (subclass or derived class) to inherit properties and behaviour s (fields and methods) from another class (superclass or base class). The subclass can reuse code from the superclass and extend or override it.
- **Purpose:** It promotes code reuse, extensibility, and the creation of a hierarchy of classes.
- **Example:**

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

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

2. Polymorphism:

- **Definition:** Polymorphism is the ability of objects of different classes to be treated as objects of a common base class. It allows methods to be written to handle objects of a generic type and work with instances of any class that extends or implements that type.
- **Purpose:** It enables flexibility in code design by allowing the same method or interface to be used for different types of objects.
- **Example:**

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

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

void letAnimalMakeSound(Animal animal) {
    animal.makeSound();
}
```

```
public static void main(String[] args) {
    Animal myDog = new Dog();
    letAnimalMakeSound(myDog);
}
```

In this example, '**letAnimalMakeSound'** can accept any object of type '**Animal'**, including instances of '**Dog'**. This is an example of polymorphism through method overriding.

In summary:

- **Inheritance** focuses on the relationship between classes, allowing a class to reuse code from another class.
- **Polymorphism** focuses on the ability of different classes to be treated as instances of a common base class, allowing methods to work with objects of various types through a shared interface.

6. How composition differs from inheritance in OOP? Explain with examples.

Composition and inheritance are two mechanisms in object-oriented programming (OOP) that facilitate code organization and reuse. They address code structuring in different ways, and each has its own advantages and use cases.

Composition:

- **Definition:** Composition is a design principle where a class consists of or is composed of other classes, rather than inheriting from them. It involves creating relationships between classes by including instances of other classes within one class. This is often referred to as "has-a" or "uses-a" relationship.
- **Example:**

```
class Engine {
    void start() {
        System.out.println("Engine starting");
    }
}

class Car {
    Engine engine;

    Car(Engine engine) {
        this.engine = engine;
    }

    void start() {
        System.out.println("Car starting");
        engine.start();
    }
```

```
}

public class Main {
   public static void main(String[] args) {
      Engine myEngine = new Engine();
      Car myCar = new Car(myEngine);

      myCar.start();  // This will start both the car and its engine
   }
}
```

In this example, the '**Car'** class has a composition relationship with the **'Engine'** class. The **'Car'** "has-a" relationship with an '**Engine'**, and it delegates the start functionality to the **'Engine'** class.

Inheritance:

- **Definition:** Inheritance is a mechanism where a class (subclass or derived class) can inherit properties and behaviours from another class (superclass or base class). It involves creating a hierarchy of classes where the subclass can reuse and extend the code of the superclass.
- **Example**

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

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

public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.makeSound();  // This will bark, as Dog inherits from Animal
   }
}
```

- In this example, the '**Dog'** class inherits from the **'Animal'** class. The **'Dog'** "is-a" kind of **'Animal'**, and it overrides the '**makeSound'** method to provide its own implementation.

Differences:

1. **Relationship Type:**
   - Composition: "Has-a" or "uses-a" relationship. Classes are connected by having instances of other classes.
   - Inheritance: "Is-a" relationship. Subclasses are related to superclasses by inheriting their properties and behaviour s.
2. **Code Reuse:**
   - Composition: Code reuse is achieved by creating relationships between classes and using their instances.
   - Inheritance: Code reuse is achieved by allowing a subclass to inherit and extend the code of a superclass.
3. **Flexibility:**
   - Composition: Offers more flexibility as it allows changing the behaviour dynamically by swapping components at runtime.
   - Inheritance: Can lead to a more rigid structure, and changes in superclass behaviour may affect all subclasses.
4. **Example Scenarios:**
   - Composition: Useful when you want to create flexible and reusable components that can be combined in various ways.
   - Inheritance: Useful when there is a clear "is-a" relationship between classes, and you want to model a hierarchy.

In practice, both composition and inheritance have their places, and the choice between them depends on the specific requirements of the problem at hand. Often, a combination of both techniques is used to achieve a well-structured and maintainable codebase.

7. How dynamic binding is achieved in polymorphism?

Dynamic binding, also known as late binding or runtime polymorphism, is achieved in polymorphism through a mechanism called virtual function or method dispatch. Dynamic binding allows the selection of the appropriate method implementation at runtime based on the actual type of the object rather than its declared type. This is a crucial aspect of polymorphism in object-oriented programming.

Here are the key components and steps involved in achieving dynamic binding:

1. Inheritance:

- Dynamic binding is closely associated with inheritance. In a polymorphic scenario, you have a base class (or interface) and one or more derived classes that override or implement certain methods.

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

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

2. Polymorphism (Method Overriding):

- The derived classes override (or implement) methods declared in the base class. This establishes a common interface, allowing objects of different derived classes to be treated uniformly through the base class reference or interface reference.

```
Animal myDog = new Dog();
myDog.makeSound();  // This will call the overridden method in the Dog class
```

3. Late Binding (Dynamic Binding):

- The method to be executed is determined at runtime based on the actual type of the object. This is in contrast to early binding (static binding), where the method is selected at compile-time based on the declared type of the reference variable.
- In the example above, the method '**makeSound'** is bound at runtime to the version provided by the actual object type **(Dog)**. This allows the system to invoke the correct method based on the runtime type of the object.

  In summary, dynamic binding is achieved through the combination of inheritance and method overriding. It allows for flexibility and extensibility in code by enabling different implementations of a common interface to be chosen at runtime based on the actual type of the object.

8.What is the difference between ”is-a” and ”has-a” relationship?

"Is-a" and "has-a" are phrases used in object-oriented programming to describe relationships between classes.

1. "Is-a" Relationship:

- **Definition:** In an "is-a" relationship, one class is considered to be a type of another class. This relationship usually involves inheritance, where a subclass inherits from a superclass.
- **Example:** If class B inherits from class A, we can say that "B is a type of A."
- **Illustration:**

```
class Animal { /* ... */ }
class Dog extends Animal { /* ... */ }
```

Usage: "Is-a" relationships are commonly expressed using inheritance, and they are used to model a hierarchy where a subclass is a specialized version of its superclass.

2. "Has-a" Relationship:

- **Definition:** In a "has-a" relationship, one class has another class as a component or member. This relationship is typically implemented through composition, where an object of one class is part of another class.
- **Example:** If class B has an instance of class A as a member, we can say that "B has an A."
- **Illustration:**

```
class Car {
   Engine engine; // Car has an Engine
}
class Engine { /* ... */ }
```

- **Usage:** "Has-a" relationships are commonly expressed using composition, and they are used to model a scenario where one class contains or uses another class as a component.

Summary:

- **"Is-a" Relationship:**
  - Involves inheritance.
  - Represents a type/subtype or generalization/specialization relationship.
  - Examples: '**Dog'** is a type of **'Animal'**, **'Circle'** is a type of **'Shape'**.
- **"Has-a" Relationship:**
  - Involves composition.
  - Represents a containment or ownership relationship.
  - Examples: **'Car'** has an **'Engine'**, **'Person'** has an **'Address'**.

9. Explain polymorphism briefly. Write down the roles of various types of polymorphism.

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their base class or interface, providing a unified interface for different types. Polymorphism enables flexibility, extensibility, and code reuse by allowing methods to work with objects of various types through a shared interface.

There are two main types of polymorphism: compile-time (or static) polymorphism and runtime (or dynamic) polymorphism.

1. **Compile-time Polymorphism (Method Overloading):**

- **Role:** Compile-time polymorphism is achieved through method overloading, where multiple methods in the same class have the same name but different parameter lists (different number or types of parameters).

- **Example:**

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

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

2. Runtime Polymorphism (Method Overriding):

- **Role:** Runtime polymorphism is achieved through method overriding, where a subclass provides a specific implementation for a method that is already defined in its superclass. It is also known as late binding or dynamic polymorphism.

- **Example:**

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

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

- **Usage:** Achieved when a method is called on a reference to a base class (or interface) that is pointing to an object of a derived class. The actual method to be executed is determined at runtime based on the type of the actual object.

3. Parametric Polymorphism (Generics):

- **Role:** Parametric polymorphism, commonly known as generics in Java, allows the creation of classes, interfaces, and methods that operate on types parameterized by other types. It enhances code flexibility and type safety.
- **Example:**

```
class Box<T> {
   private T value;
   void setValue(T value) {
      this.value = value;
   }
   T getValue() {
      return value;
   } }
```

- **Usage:** Enables writing code that can work with different types without sacrificing type safety.

Polymorphism, in general, plays a crucial role in achieving flexibility, modularity, and maintainability in software systems. It allows the development of code that can adapt to different scenarios and encourages the creation of reusable and extensible components.

10. How can we achieve dynamic polymorphism briefly? Explain with example.

Dynamic polymorphism in Java is achieved through method overriding, also known as late binding. Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The actual method to be executed is determined at runtime based on the type of the actual object.

Here is a brief explanation and an example of dynamic polymorphism:

1. **Method Overriding:**
   - **Definition:** In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass should have the same signature (name, return type, and parameters) as the method in the superclass.
   - **Key Points:**
   - The method in the subclass must be marked with the @Override annotation to ensure that it is intended to override a superclass method.
   - The access level of the overriding method cannot be more restrictive than the access level of the overridden method in the superclass.
2. **Example:**
   - Consider a scenario where you have a base class **'Animal'** with a method '**makeSound'**. The '**Dog'** class, which is a subclass of **'Animal'**, overrides the '**makeSound'** method to provide a specific implementation for a dog's sound.

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

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

Now, you can create an instance of **'Dog'** and call the **'makeSound'** method. At runtime, the JVM determines the actual type of the object **('Dog')**, and it invokes the overridden method in the **'Dog'** class.

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

In this example, '**myDog'** is declared as type '**Animal'**, but at runtime, it refers to an instance of Dog. When the '**makeSound'** method is called on '**myDog'**, it dynamically binds to the overridden method in the '**Dog'** class, resulting in the output "**Bark**." This is an illustration of dynamic polymorphism in action.

11. What are the types of Polymorphism?

There are two main types of polymorphism in object-oriented programming: compile-time polymorphism (also known as static polymorphism) and runtime polymorphism (also known as dynamic polymorphism).

1. **Compile-time Polymorphism (Static Polymorphism):**
   - **Method Overloading:**
   - **Definition:** Method overloading is a form of compile-time polymorphism where multiple methods in the same class have the same name but different parameter lists (different number or types of parameters).
   - **Example:**

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

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

Operator Overloading:

- **Definition:** Operator overloading is a form of compile-time polymorphism where the behaviour of an operator is defined for user-defined types.
- **Example:**

```
class ComplexNumber {
double real;
double imaginary;
```

```
    ComplexNumber add(ComplexNumber other) {
    ComplexNumber result = new ComplexNumber();
    result.real = this.real + other.real;
    result.imaginary = this.imaginary + other.imaginary;
    return result;
   }
}
```

2**. Runtime Polymorphism (Dynamic Polymorphism):**

- **Method Overriding:**
  - **Definition:** Method overriding is a form of runtime polymorphism where a subclass provides a specific implementation for a method that is already defined in its superclass. The actual method to be executed is determined at runtime based on the type of the actual object.
  - **Example:**

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

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

Interface Polymorphism:

- **Definition:** Interface polymorphism is a form of runtime polymorphism where objects of different classes that implement the same interface can be treated uniformly through the interface reference.
- **Example:**

```
interface Shape {
   void draw();
}
class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a circle");
   }
}
class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a rectangle");     } }
```

In summary, polymorphism in Java includes both compile-time polymorphism (method overloading, operator overloading) and runtime polymorphism (method overriding, interface polymorphism). Each type serves different purposes and contributes to the flexibility and extensibility of object-oriented code.

12. Explain Compile-time Polymorphism with example.

Compile-time polymorphism in Java is achieved through method overloading. Method overloading allows multiple methods in the same class to have the same name but different parameter lists. The appropriate method to be called is determined at compile-time based on the number and types of arguments passed to the method. This is also known as static polymorphism or early binding.

Here's an example of compile-time polymorphism using method overloading:

```
class MathOperations {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    double add(double a, double b) {
        return a + b;
    }

    // Method to concatenate two strings
    String add(String a, String b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations mathOps = new MathOperations();

        // Call the add method with integers
        int sumInt = mathOps.add(5, 10);
        System.out.println("Sum of integers: " + sumInt);

        // Call the add method with doubles
        double sumDouble = mathOps.add(3.5, 7.2);
        System.out.println("Sum of doubles: " + sumDouble);

        // Call the add method with strings
        String concatenatedString = mathOps.add("Hello", " World!");
        System.out.println("Concatenated string: " + concatenatedString);
    } }
```

In this example:

1. The '**MathOperations'** class has three overloaded '**add'** methods:
    - One that adds two integers ('**int add(int a, int b)'**).
    - One that adds two doubles (**'double add(double a, double b)**').
    - One that concatenates two strings (**'String add(String a, String b)**').
2. In the '**Main'** class, an instance of **'MathOperations'** is created **(mathOps)**.
3. The **'add'** method is called three times with different types of arguments:
    - First, with integers (**'5'** and **'10'**).
    - Second, with doubles (**'3.5'** and **'7.2**').
    - Third, with strings (**"Hello"** and **" World!"**).
4. The appropriate **'add'** method is selected at compile-time based on the types of arguments passed to it. The method signature (name and parameter types) determines which version of the method is called.
5. The output demonstrates the results of the method calls, showing how method overloading allows the same method name to handle different types of input parameters.

13. Explain overloading unary and binary operators with suitable examples.

Operator overloading in Java allows you to define how operators behave for objects of user-defined types. In Java, some operators can be overloaded, including unary and binary operators. Let's explore overloading unary and binary operators with suitable examples:

Overloading Unary Operators:

Unary operators operate on a single operand. Examples of unary operators include '+', '-', '++', '--', and '!'. To overload a unary operator, you need to define a method with the appropriate name and signature.

Example: Overloading the Unary Minus (-) Operator:

```
class MyNumber {
    private int value;

    public MyNumber(int value) {
        this.value = value;
    }

    // Overloading the unary minus operator
    public MyNumber unaryMinus() {
        return new MyNumber(-value);
    }

    public int getValue() {
        return value;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MyNumber number = new MyNumber(5);

        // Using the overloaded unary minus operator
        MyNumber negatedNumber = number.unaryMinus();

        System.out.println("Original Number: " + number.getValue());
        System.out.println("Negated Number: " + negatedNumber.getValue());
    }
}
```

In this example, the **'MyNumber'** class overloads the unary minus operator ('-') by defining a method named **'unaryMinus'**. When this method is called, it returns a new **MyNumber** object with the negated value.

Overloading Binary Operators:

Binary operators operate on two operands. Examples of binary operators include '+', '-', '*,' /', '%', '==', '!=', '>', '<', etc. To overload a binary operator, you need to define a method with the appropriate name and signature.

Example: Overloading the Binary Addition (+) Operator:

```
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Overloading the binary addition operator
    public Point add(Point other) {
        return new Point(this.x + other.x, this.y + other.y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Point point1 = new Point(2, 3);
        Point point2 = new Point(5, 7);

        // Using the overloaded binary addition operator
        Point result = point1.add(point2);

        System.out.println("Point 1: (" + point1.getX() + ", " + point1.getY() + ")");
        System.out.println("Point 2: (" + point2.getX() + ", " + point2.getY() + ")");
        System.out.println("Result: (" + result.getX() + ", " + result.getY() + ")");
    }
}
```

In this example, the **'Point'** class overloads the binary addition operator ('+') by defining a method named **'add'**. When this method is called, it returns a new **'Point'** object with coordinates obtained by adding the corresponding coordinates of two points.

These examples demonstrate how to overload unary and binary operators in Java, providing custom behaviour for operations on user-defined types.

14. What are the differences between 'this' and 'super' keyword?

The **'this**' and **'super**' keywords are both used in Java to refer to objects, but they have different meanings and are used in different contexts. Here are the main differences between **'this**' and **'super**':

1. **Purpose:**
   - **this:** It is a reference variable in Java that refers to the current object. It is primarily used to differentiate instance variables from local variables when they have the same name, and it is also used to invoke current object's method.
   - **super:** It is a reference variable that is used to refer to the immediate parent class object. It is used to access members (fields and methods) of the superclass, and it is often used to invoke the superclass's version of overridden methods.
2. **Usage:**
   - **this:**
     - Used to differentiate instance variables from local variables with the same name.
     - Used to invoke the current object's method.
     - Used in constructors to invoke another constructor in the same class.
   - **super:**
     - Used to access members of the superclass when they are hidden or overridden in the subclass.
     - Used to invoke the superclass's constructor.
3. **Scope:**

- this: Refers to the current instance of the class. It is primarily used within the scope of the instance methods or constructors of a class.
- super: Refers to the superclass of the current class. It can be used in the context of instance methods, constructors, or even in the context of a subclass's static method to call a superclass's static method.

4. **Example:**

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    String breed;

    Dog(String name, String breed) {
        super(name); // Using 'super' to invoke superclass constructor
        this.breed = breed;
    }

    void bark() {
        System.out.println("Dog is barking");
    }

    void displayDetails() {
        System.out.println("Name: " + super.name); // Using 'super' to access
    superclass field
        super.eat(); // Using 'super' to invoke superclass method
        this.bark(); // Using 'this' to invoke current class method
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever");
        myDog.displayDetails();
    }
}
```

In this example, **'this'** is used to differentiate between the instance variable **'name'** and the parameter **'name**' in the constructor of the '**Animal'** class. **'super**' is used to access the name field of the superclass **'Animal'** and to invoke the **'eat'** method of the superclass. The **'this'** keyword is used to invoke the **'bark'** method of the current class.

In summary, '**this'** and '**super'** have distinct roles in Java:

'this' refers to the current object and is used to differentiate instance variables from local variables and to invoke the current object's methods.

'super' refers to the superclass object and is used to access members of the superclass and to invoke the superclass's constructor and methods.

15. What are the rules to be followed while overriding a method?

When overriding a method in Java, there are certain rules and conventions that should be followed to ensure proper functioning of the program and to adhere to the principles of polymorphism. Here are the key rules for method overriding:

Method Signature:

The method in the subclass must have the same method signature (name, return type, and parameters) as the method in the superclass.

Access Level:

The access level of the overriding method in the subclass cannot be more restrictive than the access level of the overridden method in the superclass. It can be the same or less restrictive, but not more restrictive.

Return Type:

The return type of the overriding method in the subclass must be the same as, or a subtype of, the return type of the overridden method in the superclass. Covariant return types were introduced in Java 5.

Exception Handling:

If the overridden method in the superclass declares checked exceptions, the overriding method in the subclass can declare the same, a subclass, or no exceptions. However, it cannot declare broader checked exceptions.

Final and Static Methods:

A final method in the superclass cannot be overridden in the subclass. Similarly, a static method in the superclass cannot be overridden in the subclass, as static methods are resolved at compile-time, not runtime.

Abstract Methods:

If a subclass is not an abstract class, it must provide concrete implementations for all abstract methods inherited from its abstract superclass. Failure to do so will result in a compilation error.

Use of '@Override' Annotation:

While it is not strictly required, it is a good practice to use the @Override annotation when overriding a method. This annotation informs the compiler that the intention is to override a method, and it helps catch errors if the method signature does not match any method in the superclass.

Here's an example illustrating method overriding with adherence to the rules:

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

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

In this example, the **'makeSound'** method in the '**Dog'** class overrides the **'makeSound'** method in the **'Animal**' class. The method signature, access level, and return type are consistent with the rules for method overriding.

16. Explain public, private and protected access specifiers and show their visibility when they are inherited as public, private and protected.

In Java, access specifiers determine the visibility and accessibility of classes, methods, and fields. There are three main access specifiers: public, private, and protected. Additionally, there is a default (package-private) access level, which is the default when no access specifier is explicitly specified.

Here's an explanation of each access specifier:

1. **'public':**
   - **Visibility:** Members (classes, methods, fields) with public access specifier are visible and accessible from any other class.

- **Inheritance Visibility:**
- Public members of a superclass remain public when inherited by a subclass.
- Public members can be accessed by any subclass or any other class in the same or different package.

```
public class A {
    public int publicField;
    public void publicMethod() {
        // Code here
    }
}
```

2. **'private':**

- **Visibility:** Members with private access specifier are only visible and accessible within the same class.
- **Inheritance Visibility:**
    - Private members of a superclass are not directly accessible in a subclass.
    - They are not inherited in the usual sense, and attempts to access them in the subclass result in a compilation error.

```
public class A {
    private int privateField;
    private void privateMethod() {
        // Code here
    }
}
```

3. **protected:**

Visibility: Members with protected access specifier are visible and accessible within the same package and by subclasses, regardless of the package.

- **Inheritance Visibility:**
    - Protected members of a superclass remain protected when inherited by a subclass.
    - They are visible to subclasses and other classes in the same package.

```
public class A {
    protected int protectedField;
    protected void protectedMethod() {
        // Code here
    }
}
```

Visibility When Inherited:

Let's consider a scenario where class **'B'** extends class **'A'**:

```
public class B extends A {
   void exampleMethod() {
      // Accessing inherited members
      int x = publicField;    // OK (public is inherited)
      int y = protectedField; // OK (protected is inherited)
      // int z = privateField; // Compilation error (private not directly accessible)
      // privateMethod();      // Compilation error (private not directly accessible)
   }
}
```

In this example, '**publicField'** and '**protectedField'** from class A are inherited by class B. '**publicField'** is accessible directly because it is **'public'**. '**protectedField'** is also accessible because '**B'** is a subclass of **'A'**, and '**protected'** members are accessible to subclasses.

It's important to note that while protected members are visible to subclasses, they are also visible within the same package. If the classes **'A'** and '**B**' were in different packages, '**protectedField'** would only be accessible in the subclass '**B**' due to inheritance, not due to package visibility.

17. What do you mean by constructors in derived classes? If a constructor function is defined for derived and base class, then which constructor function gets executed first, explain with example.

In object-oriented programming, a constructor is a special method that is called when an object is instantiated. Constructors are used to initialize the state of an object, and they are typically defined in the class in which the object is being created. When dealing with inheritance, the concept of constructors in derived (subclass) and base (superclass) classes comes into play.

Constructors in Derived Classes:

In a derived class, you can define a constructor to initialize the additional attributes introduced by the derived class. The derived class constructor typically starts with a call to the constructor of the base class using the **super()** keyword, ensuring that the initialization of the base class attributes is performed.

Constructor Execution Order:

When an object of a derived class is created, the constructor of the base class is executed first, followed by the constructor of the derived class. This ensures that the initialization of the base class is completed before the derived class adds its own initialization logic.

Example:

Consider the following example with a base class **Animal** and a derived class **Dog**:

```
class Animal {
    private String species;

    // Base class constructor
    public Animal(String species) {
        this.species = species;
        System.out.println("Animal constructor called");
    }

    public void displaySpecies() {
        System.out.println("Species: " + species);
    }
}

class Dog extends Animal {
    private String breed;

    // Derived class constructor
    public Dog(String species, String breed) {
        super(species); // Call to the base class constructor
        this.breed = breed;
        System.out.println("Dog constructor called");
    }

    public void displayBreed() {
        System.out.println("Breed: " + breed);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Canine", "Labrador");
        myDog.displaySpecies();
        myDog.displayBreed();
    }
}
```

In this example:

- The '**Animal'** class has a constructor that initializes the '**species'** attribute.
- The '**Dog'** class extends '**Animal'** and has its own constructor that initializes the '**breed'** attribute. It uses '**super(species)'** to call the constructor of the base class (**'Animal'**).
- When creating a **'Dog'** object '**(myDog)'** in the **'Main**' class, the sequence of constructor calls is as follows:
    1. The '**Animal'** constructor is called first.
    2. Then, the **'Dog'** constructor is called.

The output of the program would be:

Animal **constructor** called
Dog **constructor** called
Species: Canine
Breed: Labrador

This demonstrates the order in which constructors are executed when dealing with inheritance. The base class constructor is called before the derived class constructor.

18. What do you mean by hybrid inheritance? Explain how to remove ambiguity in the case of hybrid inheritance?

Hybrid inheritance is a combination of two or more types of inheritance within a single program. In other words, it is a mix of different inheritance types, such as multiple inheritance, multilevel inheritance, or hierarchical inheritance, within the same program. Hybrid inheritance often involves the use of interfaces and abstract classes to achieve the desired combination of inheritance types.

Types of Inheritance involved in Hybrid Inheritance:

1. **Multiple Inheritance:** A class inherits from more than one class.
2. **Multilevel Inheritance:** A class is derived from another class, which is further derived from another class.
3. **Hierarchical Inheritance:** Multiple classes are derived from a single base class.

Example of Hybrid Inheritance:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C {
    void methodC() {
        System.out.println("Method C");
    }
}

class D extends C implements A, B {
    // Implementing methods from interfaces A and B
    public void methodA() {
        System.out.println("Method A");
    }
```

```
    public void methodB() {
        System.out.println("Method B");
    }
}

public class Main {
    public static void main(String[] args) {
        D obj = new D();
        obj.methodA();
        obj.methodB();
        obj.methodC();
    }
}
```

In this example, classb implements two interfaces **('A'** and **'B')** and extends a class **('C')**. This is an example of hybrid inheritance, combining multiple inheritance (interfaces **'A'** and **'B'**) and multilevel inheritance (class **'C'** is a base class for **'D'**).

Removing Ambiguity in Hybrid Inheritance:

Ambiguity may arise in hybrid inheritance scenarios, especially in cases where a class is inheriting from multiple classes or implementing multiple interfaces, and there is a potential conflict in method signatures. To remove ambiguity, you can follow these guidelines:

1. **In case of Method Overriding:**
   - If a class is inheriting from multiple classes and there is a conflict in method signatures, explicitly override the method in the derived class to provide a clear implementation.
2. **In case of Interface Implementation:**
   - If a class is implementing multiple interfaces and there is a conflict in method signatures, provide a concrete implementation for the conflicting methods in the implementing class.
3. **Use of super Keyword:**
   - When resolving ambiguity in method calls, you can use the **'super'** keyword to explicitly specify which superclass or interface the method belongs to.

- **Example**

```
interface A {
    void method();
}

interface B {
    void method();
}
```

```
class C implements A, B {
    // Resolving ambiguity in method implementation
    public void method() {
        System.out.println("Implementation in class C");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.method();  // Calls the method in class C
        ((A)obj).method();  // Calls the method from interface A
        ((B)obj).method();  // Calls the method from interface B
    }
}
```

In this example, the '**C'** class implements both interfaces **'A**' and **'B'**. The ambiguity is resolved by providing a concrete implementation for the **'method'** in class **'C'**. When calling the method, you can use explicit casting to choose which interface's method to call.

19. Explain different visibility modes with example.

In Java, visibility modes (also known as access modifiers) determine the visibility and accessibility of classes, methods, and fields. There are four main visibility modes in Java: **'public', 'private', 'protected',** and **'default (package-private)'**. Let's discuss each mode with examples:

1. **'public':**

- **Visibility:** Members (classes, methods, fields) with the **'public'** access modifier are visible and accessible from any other class.
- **Example**

```
public class PublicExample {
    public int publicField;

    public void publicMethod() {
        System.out.println("Public method");
    }
}
```

2. **private:**

Visibility: Members with the **'private'** access modifier are only visible and accessible within the same class.

- **Example:**

```
public class PrivateExample {
    private int privateField;

    private void privateMethod() {
        System.out.println("Private method");
    }
}
```

3. 'protected':

- **Visibility:** Members with the **'protected'** access modifier are visible and accessible within the same package and by subclasses, regardless of the package.
- **Example:**

```
public class ProtectedExample {
    protected int protectedField;

    protected void protectedMethod() {
        System.out.println("Protected method");
    }
}
```

4. 'Default (Package-Private)':

- **Visibility:** If no access modifier is specified (default), the member is package-private, which means it is visible and accessible within the same package.
- **Example:**

```
class DefaultExample {
    int packagePrivateField;

    void packagePrivateMethod() {
        System.out.println("Package-private method");
    }
}
```

Visibility in Inheritance:

When it comes to inheritance, the visibility modes also play a role in determining which members of a superclass are visible and accessible in a subclass. The rules are as follows:

- **public:** Inherited and remains public.
- **private:** Not inherited directly by subclasses.
- **protected:** Inherited and remains accessible to subclasses.
- **Default (Package-Private):** Inherited and remains accessible to subclasses within the same package.

Example with Inheritance:

```
public class BaseClass {
    public int publicField;
    private int privateField;
    protected int protectedField;
    int packagePrivateField; // Default (Package-Private)

    public void publicMethod() {
        System.out.println("Public method");
    }

    private void privateMethod() {
        System.out.println("Private method");
    }

    protected void protectedMethod() {
        System.out.println("Protected method");
    }

    void packagePrivateMethod() {
        System.out.println("Package-private method");
    }
}

public class SubClass extends BaseClass {
    void example() {
        System.out.println(publicField);       // Accessible (public)
        // System.out.println(privateField);    // Compilation error (private not
inherited)
        System.out.println(protectedField);    // Accessible (protected)
        System.out.println(packagePrivateField); // Accessible within the same
package (package-private)

        publicMethod();        // Accessible (public)
        // privateMethod();    // Compilation error (private not inherited)
        protectedMethod();     // Accessible (protected)
        packagePrivateMethod(); // Accessible within the same package (package-
private)
    }
}
```

In this example, the '**SubClass'** inherits members from the **'BaseClass**' with different access modifiers. The rules of inheritance and visibility are followed accordingly.

20. How do the properties of following two derived class differ: i. class D1: private B, public C . . . . . .; ii. class D2: protected B, private C . . . . . .;

The properties of two derived classes, '**D1'** and '**D2'**, are described based on their inheritance from classes '**B'** and '**C**' with different access specifiers. Let's analyze the properties of each derived class:

Class D1: private B , public C

```
class B {
   // ...
}

class C {
   // ...
}

class D1 extends B {
   private B objB;  // private inheritance
   public C objC;   // public inheritance

   // Other members of D1
}
```

In this case:

- **Inheritance from B:**
  - '**D1**' privately inherits from **'B'**, which means members of **'B'** are not directly accessible from outside **'D1'**.
  - D1 can contain an instance of B as a private member ('**objB'**), but it's not inheriting the members of B directly.
- **Inheritance from C:**
  - **'D1'** publicly inherits from **'C'**, which means members of **'C'** are accessible from outside **'D1'**.
  - **'D1'** can contain an instance of **'C'** as a public member ('**objC'**), and it's inheriting the members of **'C'** with public visibility.

Class D2: protected B, private C

```
class B {
   // ...
}

class C {
   // ...
}
```

```
class D2 extends B {
protected B objB;  // protected inheritance
private C objC;     // private inheritance

// Other members of D2
}
```

In this case:

Inheritance from B:

- **'D2'** protectedly inherits from **'B',** which means members of **'B'** are accessible within the class **'D2'** and its subclasses.
- **'D2'** can contain an instance of **'B'** as a protected member (**objB**), and it's inheriting the members of **'B'** with protected visibility.

Inheritance from C:

- **'D2'** privately inherits from **'C'**, which means members of **'C'** are not directly accessible from outside **'D2'** .
- **'D2'** can contain an instance of **'C'** as a private member (**objC**), but it's not inheriting the members of **'C'** directly.

Summary:

1. In '**D1'**, the inheritance from **'B'** is private, and from **'C'** is public.
   - Members of **'B'** are encapsulated within **'D1'** , and members of **'C'** are accessible from outside **'D1'**.
2. In **'D2'**, the inheritance from **'B'** is protected, and from **'C'** is private.
   - Members of **'B'** are accessible within **'D2'** and its subclasses, and members of '**C'** are encapsulated within **'D2'** .

PART-C SHORT ANSWER QUESTIONS

1.Define inheritance.

Inheritance is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes. A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass.

2. Define base class.

A base class is a class, in an object-oriented programming language, from which other classes are derived. It facilitates the creation of other classes that can reuse the code implicitly inherited from the base class (except constructors and destructors).

3. Define derived class.

In Java, as in other object-oriented programming languages, classes can be derived from other classes. The derived class (the class that is derived from another class) is called a subclass. The class from which its derived is called the superclass.

4. What is an abstract class?

An abstract class is a template definition of methods and variables in a specific class, or category of objects. In programming, objects are units of code, and each object is made into a generic class. Abstract classes are classes that contain one or more abstracted behaviours or methods.

5. What is IS-A relationship?

In Java, we have two types of relationship: Is-A relationship: Whenever one class inherits another class, it is called an IS-A relationship. Has-A relationship: Whenever an instance of one class is used in another class, it is called HAS-A relationship.

6 & 7. What is a subclass and super class?

Definitions: A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).

8. What is hybrid inheritance?

Hybrid inheritance is a combination of more than two types of inheritances single and multiple. It can be achieved through interfaces only as multiple inheritance is not supported by Java. It is basically the combination of simple, multiple, hierarchical inheritances.

9. What is hierarchical inheritance?

Hierarchical inheritance in Java is a type of inheritance in which the same class is inherited by more than one class. In other words, when several classes inherit their features from the same class, the type of inheritance is said to be hierarchical.

10. What is polymorphism?

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance. Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks.

11. What is static polymorphism?

Compile-Time Polymorphism. Also called static polymorphism, this type of polymorphism is achieved by creating multiple methods with the same name in the same class, but with each one having different numbers of parameters or parameters of different data types.

12. What is dynamic polymorphism?

Dynamic polymorphism in Java refers to the process when a call to an overridden process is resolved at the run time. The reference variable of a superclass calls the overridden method. As the name dynamic connotes, dynamic polymorphism happens among different classes as opposed to static polymorphism.

13. List out the operators which cannot be overloaded.

1. **Dot Operator (.)**
   - The dot operator is used to access members of a class (fields or methods), and it cannot be overloaded.
2. **Member Selection Operator (->)**
   - The member selection operator (->) is used in C and C++ for member access in pointers to structures. In Java, where explicit pointers are not used, this operator is not applicable.
3. **Method Call Operator (())**
   - The method call operator () is used for calling methods in Java, and it cannot be overloaded.
4. **Array Subscript Operator ([])**
   - The array subscript operator ([]) is used for accessing elements in an array, and it cannot be overloaded.
5. **Conditional Operator (?:)**
   - The conditional operator (?:) is used for ternary conditional expressions, and it cannot be overloaded.
6. ***Assignment Operators (=, +=, -=, =, /=, etc.)**
   - Assignment operators, such as =, +=, -=, *=, /=, etc., cannot be overloaded. They have a fixed meaning related to assignment and cannot be redefined.
7. **Logical Operators (&&, ||, !)**
   - Logical operators, including && (logical AND), || (logical OR), and ! (logical NOT), cannot be overloaded.
8. **Unary Operators (+, -, ++, --)**
   - Unary operators, such as + (unary plus), - (unary minus), ++ (increment), and -- (decrement), cannot be overloaded.
9. **Object Instantiation and Object Reference Operators (new, instanceof)**
   - The new operator for object instantiation and the instanceof operator for type checking cannot be overloaded.

14. Define virtual function.

A method or a function that overrides the behaviour of an inherited class function with the identical signature to accomplish polymorphism is called the virtual method or virtual function in an object-oriented programming language

15. What is a pure virtual function?

A virtual function for which we are not required implementation is considered as pure virtual function. For example, Abstract method in Java is a pure virtual function.

16. Differentiate between a class and a method.

In Java, a class is a template for creating objects. It defines the state and behaviour of objects that are created from it. A class can contain fields (variables) and methods (functions). A method is a block of code that performs a specific task and may or may not return a result.

17. Can we pass an object of a subclass to a method expecting an object of the super class?

Yes, it is possible to pass an object of a subclass to a method expecting an object of the superclass. This concept is known as polymorphism, and it is a fundamental feature of object-oriented programming. In Java, this is achieved through a mechanism called "upcasting.

18. What is the purpose of the ”super” keyword?

The super keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor. The most common use of the super keyword is to eliminate the confusion between super classes and subclasses that have methods with the same name.

19. Define virtual base class.

In object-oriented programming, a virtual base class is a nested inner class whose functions and member variables can be overridden and redefined by subclasses of an outer class. Virtual classes are analogous to virtual functions.

20. What is ”final” keyword for method?

The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override). The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...). The final keyword is called a "modifier".