ASSIGNMENT

Q1 ans-

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in Java. It allows a class (known as the subclass or derived class) to inherit the properties and behaviors of another class (known as the superclass or base class). The subclass extends the superclass, acquiring its attributes and methods, and can also provide its own specialized features.

In Java, inheritance is implemented using the `extends` keyword. The subclass extends the superclass, and it gains access to all the public and protected members (fields and methods) of the superclass, except for constructors. The subclass can then modify or extend the inherited members and add its own unique members.

Key points about inheritance in Java:

1. **Syntax of Inheritance:**

   To declare inheritance in Java, use the following syntax:

   ```
   class Subclass extends Superclass {
       // Subclass members
   }
   ```

2. **Access Modifiers:**

   The subclass can access the public and protected members of the superclass directly. However, it cannot access private members of the superclass. Members with default access (no access modifier specified) are also accessible within the same package.

3. **Single Inheritance:**

Java supports single inheritance, meaning a class can inherit from only one superclass. This helps maintain simplicity and avoids the "diamond problem" that can occur in multiple inheritance.

4. **Hierarchical Inheritance:**

Multiple subclasses can inherit from the same superclass, creating a hierarchical inheritance structure.

5. **Constructor Chaining:**

The constructors in the subclass must call the constructors of the superclass using `super()` to ensure proper initialization of inherited members.

Example of inheritance in Java:

```java
// Superclass
class Vehicle {
    protected String brand;

    public void start() {
        System.out.println("Vehicle starting...");
    }
}

// Subclass inheriting from Vehicle
class Car extends Vehicle {
    private int numWheels;

    // Subclass constructor, calling superclass constructor
```

```
    public Car(String brand, int numWheels) {
        super(); // Calls the constructor of the superclass (optional if no-arg constructor in superclass)
        this.brand = brand;
        this.numWheels = numWheels;
    }

    public void drive() {
        System.out.println("Car driving...");
    }
}
```

In this example, the class `Car` inherits from the class `Vehicle`. The subclass `Car` gains access to the `brand` field and the `start()` method of the superclass `Vehicle`. The subclass then adds its own member, the `numWheels` field, and the `drive()` method.

Q2 ans-

In Java, a superclass (also known as a base class or parent class) is a class that is extended or inherited by another class called a subclass (also known as a derived class or child class). The superclass contains common attributes and behaviors that can be shared among multiple subclasses, while the subclasses can add additional attributes and behaviors specific to their individual requirements.

Key points about superclass and subclass:

- **Superclass:**
- A superclass is a class that is defined before other classes (subclasses) that extend it.
- It is the class that is being inherited from, and it serves as a blueprint for creating objects.
- A superclass can have instance variables, methods, constructors, and other class members.

- It can be abstract or concrete (non-abstract). Abstract classes cannot be instantiated and are used to define common behaviors that subclasses must implement.
- **Subclass:**
- A subclass is a class that extends or inherits from a superclass.
- It inherits all the members (fields and methods) of the superclass and can add its own additional members.
- Subclasses can override methods of the superclass to provide specific implementations (polymorphism).

Q3 ans-

I In Java, inheritance is achieved using the `extends` keyword. To implement inheritance, you define a new class (subclass) that extends an existing class (superclass). The subclass inherits all the non-private members (fields and methods) from the superclass, allowing you to reuse and extend the functionality of the superclass.

Here's the syntax for implementing inheritance in Java:

```java
class Superclass {
    // Superclass members (fields and methods)
}

class Subclass extends Superclass {
    // Subclass members (fields and methods)
}
```

Key points about inheritance implementation in Java:

1. **Subclass Extends Superclass:** The `extends` keyword is used to declare that a class is a subclass of another class. The subclass inherits all the accessible members (public and protected) from the superclass.

2. **Access Modifiers:** Members with the `public` or `protected` access modifiers in the superclass are directly accessible in the subclass. `private` members are not accessible in the subclass.

3. **Constructor Chaining:** Constructors in the subclass must call the constructor of the superclass using `super()` to ensure proper initialization of inherited members.

4. **Overriding:** Subclasses can override superclass methods to provide their own specific implementations. This is achieved by using the `@Override` annotation.

Example of inheritance implementation in Java:

```java
// Superclass
class Animal {
    protected String species;

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

    public void makeSound() {
        System.out.println("Animal making a sound...");
```

```
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    private String breed;

    public Dog(String species, String breed) {
        super(species); // Call the constructor of the superclass (Animal)
        this.breed = breed;
    }

    @Override
    public void makeSound() {
        System.out.println("Dog barking...");
    }
}
```

In this example, the class `Dog` is a subclass of `Animal`. It inherits the `species` field and the `makeSound()` method from the `Animal` superclass. The subclass adds its own field `breed` and overrides the `makeSound()` method with a specific implementation for dogs.

Q4 ans-

Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP) and is a crucial concept in Java. It allows a single method or function to work with different

types of objects in a flexible and dynamic manner. Polymorphism enables code to be written in a generic and reusable way, leading to more maintainable and extensible software.

There are two types of polymorphism in Java:

1. **Compile-Time Polymorphism (Static Polymorphism):**

   - Also known as method overloading.

   - Occurs at compile time, when the method call is resolved based on the method signature and argument types.

   - In method overloading, multiple methods in the same class have the same name but different parameter lists.

Example of compile-time polymorphism (method overloading):

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

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

In this example, the `MathOperations` class has two `add` methods that have the same name but different parameter types. The correct method is selected at compile time based on the argument types used during the method call.

2. **Run-Time Polymorphism (Dynamic Polymorphism):**

   - Also known as method overriding.

   - Occurs at runtime, when the method call is resolved based on the actual object's type rather than the reference type.

   - In method overriding, a subclass provides a specific implementation of a method that is already defined in its superclass. The method signature in the subclass must exactly match the method in the superclass, including the method name, return type, and parameter list.

Example of run-time polymorphism (method overriding):

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

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

In this example, the `Dog` class overrides the `makeSound()` method from the `Animal` superclass. The method is resolved at runtime based on the actual type of the object, allowing

the correct `makeSound()` method to be executed based on the specific type of `Animal` (in this case, a `Dog`).

Polymorphism is a powerful concept that allows for code flexibility, reusability, and extensibility. It enables the development of more generic and maintainable code, as methods can be written to work with broader types and still operate correctly with specific implementations provided by subclasses.

Q5 ans-

Method overloading and method overriding are both concepts related to polymorphism in Java, but they serve different purposes and occur at different stages in the program's execution:

1. **Method Overloading:**

- Method overloading is a form of compile-time polymorphism, also known as static polymorphism.

- It occurs when multiple methods in the same class have the same name but different parameter lists (number or types of parameters).

- The overloaded methods must have the same name but can have different return types, access modifiers, and exceptions thrown.

- The selection of the appropriate method to call is done by the compiler based on the method signature and the arguments passed during the method call.

- Method overloading allows a class to provide multiple ways of performing the same operation, but with different types of input.

Example of method overloading:

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

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

```
    }

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

2. **Method Overriding:**

   - Method overriding is a form of run-time polymorphism, also known as dynamic polymorphism.

   - It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

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

   - The selection of the appropriate method to call is done at runtime, based on the actual type of the object that the reference points to (dynamic binding).

   - Method overriding allows a subclass to provide its own behavior for an inherited method, enabling specialization and customization of behavior.

Example of method overriding:

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

class Dog extends Animal {
    @Override
```

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

- In this example, the `Dog` class overrides the `makeSound()` method from the `Animal` superclass, providing its own specific implementation of the method.

In summary, method overloading involves having multiple methods with the same name but different parameter lists in the same class (static polymorphism), while method overriding involves a subclass providing its own implementation for a method already defined in its superclass (dynamic polymorphism). Both concepts contribute to the flexibility and extensibility of Java programs.

Q6 ans-

Abstraction is one of the fundamental principles of Object-Oriented Programming (OOP) and a key concept in Java. It involves simplifying complex systems by representing only the essential features while hiding the unnecessary details. Abstraction allows you to focus on what an object does rather than how it does it, making the code more manageable and easier to understand.

In Java, abstraction is achieved through abstract classes and interfaces. An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) that are meant to be implemented by its subclasses. An interface, on the other hand, is a blueprint of a class that defines a set of abstract methods that must be implemented by any class that implements the interface.

Key points about abstraction in Java:

- **Hiding Implementation Details:** Abstraction allows you to hide the internal implementation details of a class from its external users. Users only need to know the public interface (abstract methods and data) to interact with the object, while the implementation is kept hidden.

- **Abstract Classes:** An abstract class is a class that cannot be instantiated directly and may have one or more abstract methods. It serves as a blueprint for other classes to extend and implement.
- **Abstract Methods:** Abstract methods are methods without a body (no implementation). They are meant to be overridden and implemented by subclasses that extend the abstract class.
- **Interfaces:** An interface defines a contract that classes must adhere to by implementing all the methods declared in the interface. It provides a way to achieve multiple inheritance through interface implementation.
- **Creating Abstractions:** Abstractions represent essential features and behaviors of an object, while hiding the underlying complexity. This allows developers to work with higher-level concepts and promotes code reusability and flexibility.

Q7 ans-

The main differences between abstract methods and final methods in Java are related to their purpose, implementation, and behavior:

1. **Abstract Method:**

- An abstract method is a method declared in an abstract class or interface that does not have any implementation (i.e., no method body).

- It is meant to be overridden and implemented by concrete subclasses that extend the abstract class or implement the interface.

- Abstract methods are used to define a contract that concrete subclasses must adhere to, ensuring that all subclasses provide their specific implementations.

- The abstract keyword is used to declare an abstract method.

Example of an abstract method:

```java
abstract class Shape {
    public abstract double calculateArea();
```

}

```

2. **Final Method:**

- A final method is a method in a class that cannot be overridden or modified by any subclasses. It is the opposite of an abstract method, which must be overridden in subclasses.

- Once a method is marked as final, it cannot be changed or extended in any subclass.

- Final methods are used when you want to prevent subclasses from altering or extending specific behavior provided by the superclass.

- The final keyword is used to declare a final method.

Example of a final method:

```java
class Parent {
    public final void printMessage() {
        System.out.println("This method cannot be overridden.");
    }
}
```

In this example, the `printMessage()` method in the `Parent` class is marked as final, indicating that it cannot be modified or overridden by any subclasses.

In summary:

- Abstract methods are meant to be overridden by concrete subclasses, providing specific implementations. They are used in abstract classes or interfaces.
```

- Final methods cannot be overridden by subclasses and are used when you want to prevent changes to specific behavior provided by the superclass. They are used in regular (non-abstract) classes.

```java
public class Rectangle {
    private int width;
    private int height;

    // Constructor with width and height
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // Constructor with only width (square)
    public Rectangle(int width) {
        this(width, width);
    }

    // Default constructor (rectangle with default width and height)
    public Rectangle() {
        this(1, 1);
    }
}
```

In conclusion, constructor overloading allows you to provide multiple ways of initializing objects, making your classes more flexible and user-friendly. By offering different combinations of parameters, you enhance the reusability and adaptability of your code.

Q8 ans-

The main differences between abstract methods and final methods in Java are related to their purpose, implementation, and behavior:

1. **Abstract Method:**

   - An abstract method is a method declared in an abstract class or interface that does not have any implementation (i.e., no method body).

   - It is meant to be overridden and implemented by concrete subclasses that extend the abstract class or implement the interface.

   - Abstract methods are used to define a contract that concrete subclasses must adhere to, ensuring that all subclasses provide their specific implementations.

   - The abstract keyword is used to declare an abstract method.

Example of an abstract method:

```java
abstract class Shape {
    public abstract double calculateArea();
}
```

2. **Final Method:**

   - A final method is a method in a class that cannot be overridden or modified by any subclasses. It is the opposite of an abstract method, which must be overridden in subclasses.

- Once a method is marked as final, it cannot be changed or extended in any subclass.

- Final methods are used when you want to prevent subclasses from altering or extending specific behavior provided by the superclass.

- The final keyword is used to declare a final method.

Example of a final method:

```java
class Parent {
    public final void printMessage() {
        System.out.println("This method cannot be overridden.");
    }
}
```

In this example, the `printMessage()` method in the `Parent` class is marked as final, indicating that it cannot be modified or overridden by any subclasses.

In summary:

- Abstract methods are meant to be overridden by concrete subclasses, providing specific implementations. They are used in abstract classes or interfaces.

- Final methods cannot be overridden by subclasses and are used when you want to prevent changes to specific behavior provided by the superclass. They are used in regular (non-abstract) classes.

Q9 ans-

Abstraction and encapsulation are two essential concepts in object-oriented programming (OOP), and they serve different purposes in Java:

1. **Abstraction:**

- Abstraction is the process of simplifying complex systems by representing only the essential features while hiding the unnecessary details.

- It allows you to focus on what an object does rather than how it does it, making the code more manageable and easier to understand.

- In Java, abstraction is achieved through abstract classes and interfaces. An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) that are meant to be implemented by its subclasses. An interface defines a set of abstract methods that must be implemented by any class that implements the interface.

- Abstraction provides a higher-level view of objects and their behaviors, allowing developers to work with broader types and still operate correctly with specific implementations provided by subclasses.

- Abstraction helps in creating a clear separation between the interface and implementation, promoting code reusability and maintainability.

Example of abstraction in Java:

```java
// Abstract class representing a Shape
abstract class Shape {
    public abstract void draw(); // Abstract method (no implementation)
}
```

2. **Encapsulation:**

- Encapsulation is the process of bundling data (attributes) and methods (behaviors) that operate on the data within a single unit, i.e., a class. It allows you to control the access to the data, preventing direct manipulation from outside the class.

- Encapsulation helps in hiding the internal details of how an object is implemented, protecting the data from unwanted access and modification.

- In Java, encapsulation is achieved by using access modifiers (e.g., private, protected, public) to control the visibility of class members (fields and methods). By making fields private and providing

public getter and setter methods, you can control the access to the data and enforce data validation if needed.

- Encapsulation promotes data integrity and helps maintain a well-defined interface for interacting with the object.

Example of encapsulation in Java:

```java
class Student {
    private String name; // Private field
    private int age; // Private field

    public String getName() {
        return name; // Public getter method
    }

    public void setName(String name) {
        this.name = name; // Public setter method
    }

    public int getAge() {
        return age; // Public getter method
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age; // Public setter method with data validation
        }
    }
```

}

```

In this example, the `Student` class encapsulates the `name` and `age` fields by making them private and provides public getter and setter methods to control access to these fields.

In summary, abstraction focuses on representing only essential features and behaviors while hiding unnecessary details, achieved through abstract classes and interfaces. Encapsulation, on the other hand, focuses on bundling data and methods within a class and controlling access to the data, promoting data integrity and data hiding. Both concepts contribute to the overall design and maintainability of Java programs.

## Q10 ans –

The main difference between runtime polymorphism (also known as dynamic polymorphism) and compile-time polymorphism (also known as static polymorphism) in Java lies in the timing of method resolution and binding:

1. **Compile-Time Polymorphism (Static Polymorphism):**

   - Occurs during compile-time when the method call is resolved based on the method signature and argument types known at compile-time.

   - Also known as method overloading.

   - In method overloading, multiple methods in the same class have the same name but different parameter lists (number or types of parameters).

   - The selection of the appropriate method to call is done by the compiler based on the method signature and arguments passed during the method call.

   - Method overloading provides multiple methods with the same name but different behavior based on the number or types of arguments used.

Example of compile-time polymorphism (method overloading):
```

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

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

2. **Run-Time Polymorphism (Dynamic Polymorphism):**

   - Occurs during runtime when the method call is resolved based on the actual object's type rather than the reference type.

   - Also known as method overriding.

   - In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass.

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

   - The selection of the appropriate method to call is done at runtime, based on the actual type of the object that the reference points to (dynamic binding).

   - Method overriding allows a subclass to provide its own behavior for an inherited method, enabling specialization and customization of behavior.

Example of run-time polymorphism (method overriding):

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

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

In this example, the `Dog` class overrides the `makeSound()` method from the `Animal` superclass, providing its own specific implementation of the method. The actual method to be called is determined at runtime based on the object's type (dynamic binding).

In summary, compile-time polymorphism (method overloading) is determined at compile-time based on the method signature, while run-time polymorphism (method overriding) is determined at runtime based on the actual object's type (dynamic binding).