1) What is inheritance in Java?

Ans:

- Inheritance is a concept in object-oriented programming (OOP).

- It allows the creation of a new class (subclass) based on an existing class (superclass).

- The subclass inherits the properties and behaviors (methods and variables) of the superclass.
- In Java, inheritance is achieved using the `extends` keyword.

- The subclass can add new members or override inherited members.

- Inherited members include non-private members (methods and variables) of the superclass.
- Inheritance promotes code reuse, modularity, and organization.

- Polymorphism is closely related to inheritance, allowing objects of different subclasses to be treated as objects of the superclass.

- Inherited members can be accessed using the `super` keyword.

- Multiple inheritance is not allowed in Java (a class can only inherit from one superclass), but multiple levels of inheritance are supported.

- The `Object` class is the root of the class hierarchy in Java, and all classes implicitly inherit from it.

- Inheritance supports the "is-a" relationship, where a subclass is a specialized version of the superclass.

2) what is superclass and subclass?

Ans:

The terms "superclass" and "subclass" are used to describe the relationship between two classes connected through inheritance.

Superclass:

- Also known as a base class or parent class.

- It is the existing class from which a new class is derived or extended.

- The superclass provides a blueprint or template for the subclass.

- The superclass can contain fields, methods, and other members that are inherited by the subclass.

- The subclass inherits all the non-private members of the superclass.

- In Java, a class can have only one direct superclass.

Subclass:

- Also known as a derived class or child class.

- It is the new class created by extending or inheriting from a superclass.

- The subclass inherits the properties and behaviors (methods and variables) of the superclass.

- The subclass can add new members or override the inherited members.

- The subclass can have its own unique fields, methods, and behavior in addition to the inherited ones.

- In Java, a subclass can inherit from only one superclass at a time, but it can itself be a superclass for other subclasses.

3) How is inheritance implemented/achieved in Java?

Ans:

1. Define the Superclass:

   - Create a class that will serve as the superclass.
   - Declare the fields, methods, and constructors within the superclass as needed.

2. Define the Subclass:

- Create a new class that will serve as the subclass.
- Use the `extends` keyword, followed by the name of the superclass, to indicate the inheritance relationship.

3. Inherit Members:

- The subclass automatically inherits all the non-private members (fields and methods) of the superclass.
- This includes public and protected members.
- Private members are not inherited and are only accessible within the superclass itself.

4. Add New Members or Override Inherited Members:

- In the subclass, you can add new fields, methods, and constructors specific to the subclass.
- You can also override the inherited methods from the superclass to provide a different implementation in the subclass.
- Use the `@Override` annotation to indicate that a method in the subclass is intended to override a method in the superclass.

5. Access Inherited Members:

- In the subclass, you can directly access the inherited members (fields and methods) of the superclass using dot notation.
- Use the `super` keyword to explicitly reference and call superclass methods or access superclass fields from the subclass.

Example illustrating the implementation of inheritance in Java:

```
class Animal {
    protected String name;

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

    public void eat() {
        System.out.println(name + " is eating.");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
```

```
    @Override
    public void eat() {
        System.out.println(name + " is eating dog food.");
    }

    public void bark() {
        System.out.println(name + " is barking.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.eat(); // Output: Buddy is eating dog food.
        dog.bark(); // Output: Buddy is barking.
    }
}
```

4) What is polymorphism?

Ans:

- **Polymorphism** is a key concept in object-oriented programming (OOP).

- It allows objects of different classes to be treated as objects of a common superclass or interface.

- Polymorphism enables code flexibility, reusability, and extensibility.

- Polymorphism is achieved through method overriding and method overloading.

- Method Overriding:

  - Occurs when a subclass provides a specific implementation of a method already defined in its superclass.

  - The subclass method has the same name, return type, and parameter list as the superclass method.

  - The subclass method can have a more specific implementation or behavior.

  - Allows treating a subclass object as an object of the superclass, enabling dynamic method dispatch.

- Method Overloading:

  - Occurs when multiple methods with the same name but different parameter lists exist within a class.

  - The methods have different parameter types, counts, or order.

  - The overloaded methods can have different return types.

- Polymorphism allows objects to be manipulated in a generic way, regardless of their specific subclass.

- It promotes loose coupling, abstraction, and the "programming to an interface" concept.

5) Differentiate between method overloading and overriding.

Ans:

Here's a comparison between method overloading and overriding:

Method Overloading:

- Method overloading occurs within a single class.

- Multiple methods within the same class have the same name but different parameter lists (different types, different counts, or different order of parameters).

- The return type of the method does not play a role in method overloading.

- Overloaded methods provide different ways to perform similar operations or actions with different inputs.

- The compiler determines which overloaded method to invoke based on the method name and the arguments provided at the call site.

- Overloading is determined at compile-time and is also known as static or compile-time polymorphism.

Method Overriding:

- Method overriding occurs in a subclass that extends a superclass.

- A subclass provides a specific implementation of a method that is already defined in its superclass.

- The overriding method in the subclass must have the same name, return type, and parameter list as the overridden method in the superclass.

- The purpose of method overriding is to provide a specialized implementation of the inherited method in the subclass.

- The specific implementation of the method to be executed is determined dynamically at runtime based on the actual type of the object.

- Overriding is determined at runtime and is also known as dynamic polymorphism or late binding.

6) What is an abstraction explained with an example.

Ans:

- Abstraction is a concept in object-oriented programming (OOP) that focuses on representing the essential features of an object while hiding the unnecessary details.

- It provides a simplified and generalized view of objects and their behaviors.

- Abstraction helps manage complexity, improve code maintainability, and promote modularity.

- In Java, abstraction is achieved using abstract classes and interfaces.

- Abstract classes are declared using the `abstract` keyword and can have both abstract and concrete methods.

- Abstract methods are declared without a body and must be implemented by the subclasses.

- Abstract classes cannot be instantiated, but they can be used as a reference type.

- Interfaces provide a way to achieve abstraction by defining a contract of methods that implementing classes must adhere to.

- Interfaces only contain abstract method declarations (prior to Java 8), or they can also include default and static methods.

- Abstraction allows for creating generic classes or interfaces that can be specialized by concrete implementations.

Example illustrating abstraction:

```
abstract class Animal {

    public abstract void makeSound();



    public void sleep() {

        System.out.println("Animal is sleeping.");
```

```
    }
}

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

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

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
```

```
        dog.makeSound(); // Output: Dog barks.

        cat.makeSound(); // Output: Cat meows.



        dog.sleep(); // Output: Animal is sleeping.

        cat.sleep(); // Output: Animal is sleeping.

    }

}
```

In this example, the `Animal` class is an abstract class that represents a generic animal. It declares an abstract method `makeSound()` that the concrete subclasses (`Dog` and `Cat`) must implement. The `Animal` class also provides a concrete method `sleep()`, which can be directly used by the subclasses.

The `Dog` and `Cat` classes are concrete subclasses that extend the `Animal` class. They provide their specific implementation of the `makeSound()` method.

7) What is difference between an abstract method and final method in java? Explain with an example

Ans:

The difference between an abstract method and a final method in Java lies in their behavior and purpose within the class hierarchy. Here's an explanation with an example:

Abstract Method:

- An abstract method is a method declaration that does not provide an implementation in the class where it is declared.

- It is declared using the `abstract` keyword.

- Abstract methods are meant to be overridden by the subclasses to provide their specific implementation.

- Abstract methods can only exist within abstract classes or interfaces.

- Subclasses inheriting an abstract method must implement it, providing the implementation details.

- Abstract methods cannot be marked as `final` or `private`.

- They serve as a contract for the subclasses, defining a required behavior without specifying the implementation.

Final Method:

- A final method is a method that cannot be overridden by any subclasses.

- It is declared using the `final` keyword.

- Final methods are already implemented in the class where they are declared, and they cannot be changed or overridden in the subclasses.

- They are intended to be kept as-is and not modified in any derived classes.

- Final methods can be marked as `private`, `final`, or `static`.

- They are often used to enforce specific behavior in a class, preventing subclasses from modifying or altering that behavior.

Example:

```
abstract class Shape {

    public abstract void draw(); // Abstract method



    public final void display() { // Final method

        System.out.println("Displaying shape");

    }

}



class Circle extends Shape {

    @Override

    public void draw() {

        System.out.println("Drawing circle");

    }



    // Uncommenting the below code will result in a compilation error

    // as final method cannot be overridden.

    /*

    @Override

    public final void display() {
```

```
        System.out.println("Displaying circle");
    }
    */
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();    // Output: Drawing circle
        circle.display(); // Output: Displaying shape
    }
}
```

In this example, the `Shape` class is an abstract class that declares an abstract method `draw()` and a final method `display()`. The `Circle` class extends `Shape` and provides the implementation of the `draw()` method. However, attempting to override the final method `display()` in the `Circle` class will result in a compilation error.

The purpose of the abstract method `draw()` is to define a contract that subclasses must implement to draw their specific shape. On the other hand, the final method `display()` in the `Shape` class is intended to provide a fixed implementation that cannot be modified or overridden by any subclasses.

8) What is the final class in Java?

Ans:

- A final class in Java is a class that cannot be subclassed or inherited by other classes.

- Once a class is declared as final, it cannot be extended or overridden.

- Final classes are intended to remain unchanged, preserving their behavior and structure.

- Declaring a class as final prevents modifications to its functionality.

- Final classes offer efficiency benefits as the Java compiler can perform optimizations on final methods and fields.

- Final classes can help maintain security and integrity by avoiding unintended modifications through inheritance.

- Final classes are often used for utility classes that provide helper methods or constants.

- Utility classes are not intended to be extended or modified, so making them final ensures their intended usage.

9) Differentiate between abstraction and encapsulation

Ans:

The Differences between Abstraction and Encapsulation:

Abstraction:

- Abstraction is a concept that focuses on representing the essential features of an object while hiding unnecessary details.

- It allows you to create classes or interfaces that provide a simplified and generalized view of objects and their behaviours.

- Abstraction manages complexity by emphasizing what an object does rather than how it does it.

- It helps improve code maintainability, promotes modularity, and enhances code reusability.

- In Java, abstraction is achieved using abstract classes and interfaces.

- Abstract classes can contain abstract methods (without implementation) and concrete methods.

- Interfaces declare abstract methods that must be implemented by implementing classes.

Encapsulation:

- Encapsulation is a concept that involves bundling the data and methods that operate on the data within a single unit, i.e., a class.

- It promotes data hiding and protects the internal state of an object from external access or modifications.

- Encapsulation allows for information hiding, preventing direct access to internal data from outside the class.

- It provides control over the data by enforcing access through methods (getters and setters) and maintaining the integrity of the object's state.

- Encapsulation helps in achieving data abstraction, as the internal representation and implementation details are hidden.

- Access modifiers (public, private, protected) are used to control the visibility and accessibility of class members (fields and methods).

10) Differentiate between run-time and compile time polymorphism explain with an example.

Ans:

Run-time v/s compile time polymorphism

Runtime Polymorphism (Dynamic Polymorphism):

- Runtime polymorphism, also known as dynamic polymorphism or late binding, occurs when the specific implementation of a method is determined at runtime.

- It is achieved through method overriding, where a subclass provides its specific implementation of a method that is already defined in the superclass.
- The determination of which method to execute is based on the actual type of the object.

- The appropriate method is resolved dynamically at runtime, allowing objects of different subclasses to be treated as objects of the superclass.

- Runtime polymorphism enables code flexibility and extensibility, supporting polymorphic behavior.

Example - Runtime Polymorphism:

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.makeSound();  // Output: Dog barks.
        animal2.makeSound();  // Output: Cat meows.
    }
}
```

In this example, the `Animal` class is the superclass, and the `Dog` and `Cat` classes are subclasses that override the `makeSound()` method. In the `main()` method, two animal objects (`animal1` and `animal2`) are created, which are assigned to instances of the `Dog` and `Cat` classes, respectively. When the `makeSound()` method is called on these objects, the specific implementation defined in the respective subclass is executed. This demonstrates runtime

polymorphism where the behavior of the method is determined at runtime based on the actual object type.

Compile-time Polymorphism (Static Polymorphism):

- Compile-time polymorphism, also known as static polymorphism or early binding, occurs when the specific implementation of a method is determined at compile-time.

- It is achieved through method overloading, where multiple methods with the same name but different parameter lists exist within a class.

- The determination of which method to execute is based on the method name and arguments provided during compilation.

- The appropriate method is resolved statically at compile-time, based on the method signatures and the types of arguments.

- Compile-time polymorphism allows for method selection and invocation based on the static type of the object.

Example - Compile-time Polymorphism:

```
class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }

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

public class Main {
    public static void main(String[] args) {
        int sum1 = MathUtils.add(5, 3);           // Uses int version of add()
        double sum2 = MathUtils.add(2.5, 4.7);    // Uses double version of add()

        System.out.println("Sum 1: " + sum1);     // Output: Sum 1: 8
        System.out.println("Sum 2: " + sum2);     // Output: Sum 2: 7.2
    }
}
```

In this example, the `MathUtils` class contains two overloaded versions of the `add()` method—one for integer parameters and another for double parameters. In the `main()` method, the `add()` method is invoked twice with different arguments. The compiler determines which version of the method to call based on the static types of the arguments. This demonstrates compile-time polymorphism, where the method invocation is resolved during compilation.