**Inheritance, Polymorphism and Abstraction Assignment**

1.What is in Inheritance in java

Inheritance in Java is a mechanism where a new class derives attributes and methods from an existing class. The existing class is called the superclass, and the new class is called the subclass. Inheritance allows the subclass to reuse the code of the superclass, which helps in code reusability and also allows the creation of a hierarchical classification.

Here are some key points about inheritance in Java:

1. \*\*Extends keyword\*\*: In Java, inheritance is implemented using the `extends` keyword. The subclass extends the superclass to inherit its properties.

class SubClass extends SuperClass {

// Subclass code

}

```

2. \*\*Access Modifiers\*\*: Inherited members (fields and methods) can have different access modifiers in the subclass depending on the relationship between the subclass and superclass and the desired visibility.

3. \*\*Single Inheritance\*\*: Java supports single inheritance, which means a subclass can only inherit from one superclass. This helps in avoiding ambiguity and complexity in the class hierarchy.

4. \*\*Super Keyword\*\*: The `super` keyword in Java is used to refer to the superclass. It can be used to call the superclass constructor, access the superclass methods and fields, and invoke the superclass version of an overridden method.

5. \*\*Overriding Methods\*\*: Subclasses can provide a specific implementation for a method that is already defined in its superclass. This is known as method overriding. The subclass method must have the same signature (name, parameters, and return type) as the superclass method.

6. \*\*Final Classes and Methods\*\*: In Java, you can use the `final` keyword to prevent a class from being inherited or a method from being overridden.

Inheritance promotes code reuse, reduces redundancy, and helps in creating a more organized and maintainable codebase.

2. What is superclass and subclass?

In object-oriented programming, a superclass (or parent class) is a class that is extended or inherited from by another class. It provides common properties and behaviors that are shared by its subclasses.

A subclass (or child class) is a class that inherits from a superclass. It can access the properties and methods of the superclass and can also have its own additional properties and methods. Subclasses can further be subclassed to create a hierarchy of classes with increasing specialization.

Here's a simple example in Java:

**```java**

**// Superclass**

**class Animal {**

**void eat() {**

**System.out.println("Animal is eating");**

**}**

**}**

**// Subclass**

**class Dog extends Animal {**

**void bark() {**

**System.out.println("Dog is barking");**

**}**

**}**

**```**

In this example, `Animal` is the superclass, and `Dog` is the subclass. `Dog` inherits the `eat()` method from `Animal` and also has its own `bark()` method.

3. How is inheritance implemented/achieved in java?

**package** Inheritance;

**class** Vehicle

{ **public** **void** start()

{

System.***out***.println("start...");

}

**public** **void** Drive()

{

System.***out***.println("driving..");

}

**public** **void** stop()

{

System.***out***.println("stop...");

}

}

**class** Yamaha **extends** Vehicle

{

// the methods present in vehicle class will be inherited

}

**public** **class** Practice {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Yamaha y=**new** Yamaha();

y.start();

y.Drive();

y.stop();

}

}

4. What is Polymorphism?

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Types of Java Polymorphism**

In Java Polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism

## Compile-Time Polymorphism in Java

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

### **Method Overloading**

When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

**package** com.pw.java.Polymorphism;

**class** Aeroplane

{

**public** **void** takeoff()

{

System.***out***.println("aeroplane is takking off");

}

**public** **void** fly()

{

System.***out***.println("aeroplane is flying ");

}

}

**class** CargoPlane6 **extends** Aeroplane

{

**public** **void** takeoff()

{

System.***out***.println(" cargoplane aeroplane is takking off early");

}

**public** **void** fly()

{

System.***out***.println(" cargoplane aeroplane is flying at low height ");

}

}

**class** PassengerPlane6 **extends** Aeroplane

{

**public** **void** takeoff()

{

System.***out***.println("Passenger aeroplane is takking off late");

}

**public** **void** fly()

{

System.***out***.println("Passenger aeroplane is flying at mid height ");

}

}

**class** FighterPlane **extends** Aeroplane

{

**public** **void** takeoff()

{

System.***out***.println("Fighert aeroplane is takking off late");

}

**public** **void** fly()

{

System.***out***.println(" fighter aeroplane is flying at mid height ");

}

}

// this class is used to achive polymorphism ------>

**class** Airport

{

**public** **void** poly(Aeroplane ref)

{

ref.takeoff();

ref.fly();

System.***out***.println("------------------------------------");

}

}

**public** **class** LaunchPolymorphism {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

CargoPlane6 cp=**new** CargoPlane6();

PassengerPlane6 pp= **new** PassengerPlane6();

FighterPlane fp=**new** FighterPlane();

// ------------Airport class object

Airport a= **new** Airport();

a.poly(cp);

a.poly(pp);

a.poly(fp);

}

}

5. Differentiate between method overloading and overriding

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is a compile-time polymorphism. | Method overriding is a run-time polymorphism. |
| Method overloading helps to increase the readability of the program. | Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |
| It occurs within the class. | It is performed in two classes with inheritance relationships. |
| Method overloading may or may not require inheritance. | Method overriding always needs inheritance. |
| In method overloading, methods must have the same name and different signatures. | In method overriding, methods must have the same name and same signature. |
| In method overloading, the return type can or can not be the same, but we just have to change the parameter. | In method overriding, the return type must be the same or co-variant. |
| Static binding is being used for overloaded methods. | Dynamic binding is being used for overriding methods. |
| Poor Performance due to compile time polymorphism. | It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime. |
| Private and final methods can be overloaded. | Private and final methods can’t be overridden. |
| The argument list should be different while doing method overloading. | The argument list should be the same in method overriding. |

6. What is an abstraction explained with an Example?

**Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

**Java Abstract classes and Java Abstract methods**

1. An abstract class is a class that is declared with an [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method-defined abstract must always be redefined in the subclass, thus making [overriding](https://www.geeksforgeeks.org/overriding-in-java/) compulsory or making the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

**Abstraction can be achieved by two ways**

1. **By abstract keyword (0 to100%)**
2. **By interfaces (100%)**

**1)**

**abstract class Shape {**

**abstract void draw();**

**}**

**class Circle extends Shape {**

**void draw() {**

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

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Shape shape = new Circle();**

**shape.draw(); // Output: Drawing a circle**

**}**

**}**

**2)**

**interface Animal {**

**void eat();**

**void sleep();**

**}**

**class Dog implements Animal {**

**public void eat() {**

**System.out.println("Dog is eating");**

**}**

**public void sleep() {**

**System.out.println("Dog is sleeping");**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Animal dog = new Dog();**

**dog.eat(); // Output: Dog is eating**

**dog.sleep(); // Output: Dog is sleeping**

**}**

**}**

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

In Java, abstract methods and final methods serve different purposes:

1. Abstract Method:

- An abstract method is a method declared in an abstract class or interface but does not have an implementation.

- Subclasses of the abstract class or classes implementing the interface must provide an implementation for the abstract method.

- Abstract methods are meant to be overridden by subclasses to provide specific implementations.

2. Final Method:

- A final method is a method that cannot be overridden by subclasses. When a method is declared as final, it means that the method implementation is final and cannot be changed in any subclass.

- Final methods are commonly used in classes where the behavior of a method should not be changed or extended by subclasses.

Here's an example to illustrate the difference:

**```java**

**abstract class Shape {**

**abstract void draw(); // Abstract method**

**final void display() {**

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

**}**

**}**

**class Circle extends Shape {**

**void draw() {**

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

**}**

**// Cannot override display method because it is final**

**// void display() { }**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Circle circle = new Circle();**

**circle.draw(); // Output: Drawing a circle**

**circle.display(); // Output: Displaying shape**

**}**

**}**

**```**

In this example, `Shape` is an abstract class with an abstract method `draw()` and a final method `display()`. The `Circle` class extends `Shape` and provides an implementation for the `draw()` method but cannot override the `display()` method because it is final.

**8.What is final class in java?**

In Java, a final class is a class that cannot be subclassed. This means that you cannot create a subclass of a final class. Declaring a class as final prevents other classes from extending it and overriding its methods. Final classes are often used to create immutable classes or classes with final behavior that should not be changed.

Here's an example of a final class:

**final class FinalClass {**

**// Class implementation**

**}**

**```**

9.Differentiate Between abstraction and encapsulation

| **Abstraction** | **Encapsulation** |
| --- | --- |
| Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |
| Abstraction provides access to specific part of data. | Encapsulation hides data and the user can not access same directly (data hiding. |
| Abstraction focus is on “what” should be done. | Encapsulation focus is on “How” it should be done. |

10.Differentiate between Runtime and Compile time polymorphism explain with an example

| **Compile Time Polymorphism** | **Run time Polymorphism** |
| --- | --- |
| In Compile time Polymorphism, the call is resolved by the compiler. | In Run time Polymorphism, the call is not resolved by the compiler. |
| It is also known as Static binding, Early binding and overloading as well. | It is also known as Dynamic binding, Late binding and overriding as well. |
| Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type. | Method overriding is the runtime polymorphism having the same method with same parameters or signature but associated withcompared, different classes. |
| It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| It provides fast execution because the method that needs to be executed is known early at the compile time. | It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime. |
| Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |
| Inheritance is not involved. | Inheritance is involved. |

**1)compile time Polymorphism**

// compile-time polymorphism

**public** **class** GFG {

    // First addition function

**public** **static** **int** add(**int** a, **int** b)

    {

**return** a + b;

    }

    // Second addition function

**public** **static** **double** add(

**double** a, **double** b)

    {

**return** a + b;

    }

    // Driver code

**public** **static** **void** main(String args[])

    {

        // Here, the first addition

        // function is called

        System.out.println(add(2, 3));

        // Here, the second addition

        // function is called

        System.out.println(add(2.0, 3.0));

    }

}

**2)Runtime Polymorphism**

// runtime polymorphism

// Implementing a class

**class** Test {

    // Implementing a method

**public** **void** method()

    {

        System.out.println("Method 1");

    }

}

// Defining a child class

**public** **class** GFG **extends** Test {

    // Overriding the parent method

**public** **void** method()

    {

        System.out.println("Method 2");

    }

    // Driver code

**public** **static** **void** main(String args[])

    {

        Test test = **new** GFG();

        test.method();

    }

}