Name: Gowtham Raja
Department: Impact Training
Date: 21.11.2001
Task: OOPS

INHERITANCE:

- Inheritance is one of the Key features of oops that allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from where the child is derived is known as supper class (parent or base class)
- A child inherits the traits of his/her parents. With inheritance, we can reuse the field and methods of the existing class.

Types of Inheritance:

- Single Inheritance
- Multi-level Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance:

- Single Inheritance in Java refers to the concept where one class (child class) inherits the properties and behaviors (fields and methods) of another class (parent class).
- In Java, a subclass can inherit from only one superclass, which is why it's called "single" inheritance.

Parent Class (Superclass):
This is the class whose properties and methods are inherited by another class

Child Class (Subclass):
This is the class that is inherited from the parent class. It can use and override the methods and fields of the parent class.

Method Overriding:
A child class can provide its own implementation of a method that is already defined in the parent class.

Advantages of Single Inheritance:

Easier to understand:
Single inheritance is simpler to follow because a subclass inherits from only one parent class. This leads to clear hierarchies and makes the code easier to maintain and debug.

Less ambiguity:

With only one parent class, there are fewer complexities when determining which methods or attributes are inherited by the subclass.

Example:

```
 class Animal {


void sound () {
      System.out.println ("Some sound");
   }
}
class Dog extends Animal {
   void bark () {
      System.out.println ("Bark");
   }
}
```

Better Reusability:

- With single inheritance, code reuse becomes easier because the subclass directly inherits methods and attributes from a single superclass.
- This avoids redundancy and helps in reusing common functionality, making code more modular and scalable.

Performance Considerations:

- Single inheritance may result in better performance because the JVM (Java Virtual Machine) only needs to follow a single class hierarchy.
- In multiple inheritance, it may have to resolve which parent class or method to invoke, which adds overhead.
- This is especially relevant in large systems, where the added complexity of multiple inheritance could introduce performance bottlenecks.

Multilevel Inheritance:

- Multilevel inheritance is a type of inheritance where a class is derived from another class, which is also derived from another class, and so on.
- In other words, it involves a chain of inheritance where a class is derived from another class, which in turn is derived from another class, creating a hierarchy.
- Multilevel inheritance is supported, and it allows a class to inherit the properties and behaviors (methods) of more than one class by chaining them in a sequence.
- However, each class in the chain has only one direct parent, meaning that Java does not support multiple inheritance of classes.

Example:

```
 class Animal {
   void eat () {

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

```
    }
}
class Mammal extends Animal {
    void sleep() {
        System.out.println("Mammal is sleeping.");
    }

}
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}
public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();

        dog.eat();
        dog.sleep();
        dog.bark();
    }
}
```

Multiple Inheritance in Java:

- In object-oriented programming (OOP), multiple inheritance refers to a feature where a class can inherit from more than one class. This is a common concept in languages like C++ but is not supported in Java directly for classes.
- Multiple inheritance through interfaces. This means a class can implement multiple interfaces, thus allowing it to inherit behavior from more than one source without the issues associated with multiple inheritance in classes.

Example:

```
interface Interface1 {
    void method1();
}

interface Interface2 {
    void method2();
}

class ConcreteClass implements Interface1, Interface2 {

public void method1() {
        System.out.println("Method 1 from Interface1");
  }
    public void method2() {
        System.out.println("Method 2 from Interface2");
    }
}
```

Hierarchical Inheritance:

- Hierarchical inheritance in Java is a type of inheritance where multiple classes inherit from a single parent class
- This structure creates a hierarchy, where the parent class provides common functionality to all the child classes, while each child class can add its specific functionality.

Example:

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

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

class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing.");
    }
}

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

        dog.eat();

dog.sleep();
```

```
        dog.bark();

        cat.eat();
        cat.sleep();
        cat.meow();
    }
}
```

Hybrid Inheritance:

- Hybrid inheritance in Java refers to a combination of two or more types of inheritance in a single program. In this context, hybrid inheritance typically refers to the combination of single inheritance, multiple inheritance (through interfaces), and multilevel inheritance.
- Thus, hybrid inheritance in Java often involves using interfaces alongside other forms of inheritance.
- A class can extend to another class (single inheritance) and also implement multiple interfaces (multiple inheritance via interfaces).
- You can have multilevel inheritance with interfaces involved at different levels.

Example:

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

interface Movable {
    void move();
}

interface Flyable {
    void fly();
}

class Dog extends Animal implements Movable, Flyable {
    public void move() {
        System.out.println("Dog is moving");
    }

    public void fly() {
        System.out.println("Dog is flying");
    }

}
```

Name: Gowtham Raja   Department: Impact Training
Date: 21.11.2001   Task: OOPS

```
public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.eat();    // Inherited from Animal
      dog.move();
      dog.fly();
   }
}
```

Static Keyword:

- The static keyword in Java is used to indicate that a member (variable, method, block, or nested class) belongs to the class rather than instances (objects) of the class.
- This means that the static member is shared by all instances of the class, and you can access it directly using the class name without creating an object.

Static Variables (Class Variables):
A static variable is shared by all instances of the class. It is initialized only once when the class is loaded into memory.

Static Methods:
A static method can be called without creating an object of the class. Static methods can access only static variables and other static methods.

Example:

```
      class Counter {
   static int count = 0;

   Counter() {
      count++;
   }

   void displayCount() {
      System.out.println("Current count: " + count);
   }
}

public class Main {
   public static void main(String[] args) {
      Counter obj1 = new Counter();
      Counter obj2 = new Counter();
      Counter obj3 = new Counter();


      obj1.displayCount();
```

```
        obj2.displayCount();
        obj3.displayCount();
  }
}
```

Final Keyword:

```
                class Example {
    final int number = 100;

    void changeNumber() {
        // number = 200;
        System.out.println("Final variable value: " + number);

}
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();
        obj.changeNumber();
        // obj.number = 200;
    } }
```

- Final Class is marked as final, meaning it cannot be extended or subclassed.
- If you attempt to create a subclass (e.g., Subclass extends final class), it will result in a compilation error.