Week3-Session 1- Facilitation Guide

INDEX

For (2 hrs) ILT

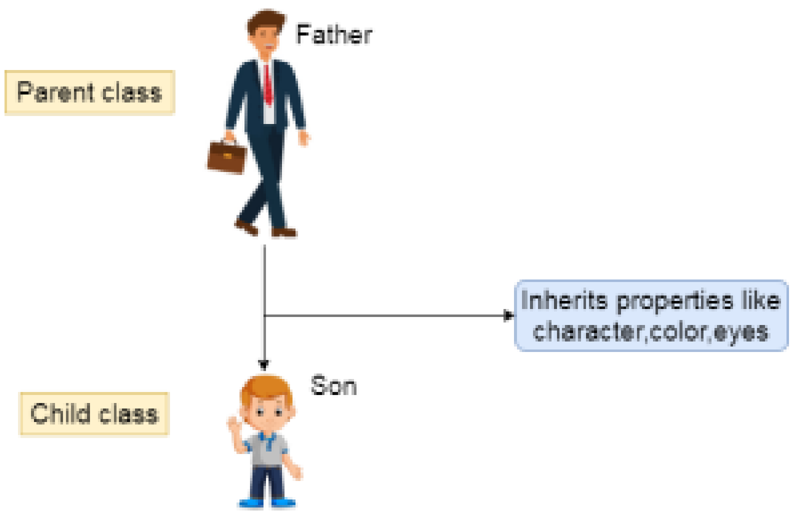
I. Recap of the previous session

In the last session we discussed wrapper classes in Java and explained how wrapper classes made Java more object-oriented. It allows us to create objects from primitive data. We discussed many techniques to convert primitive to wrapper class and vice versa. For example, we can convert an int to an Integer object and on the other hand get back the primitive value from the Integer object . We also explained with examples the benefit of using wrapper class over primitive data types.

In this session we will explore the concept of inheritance in detail. We will also learn about runtime polymorphism implemented through method overriding.

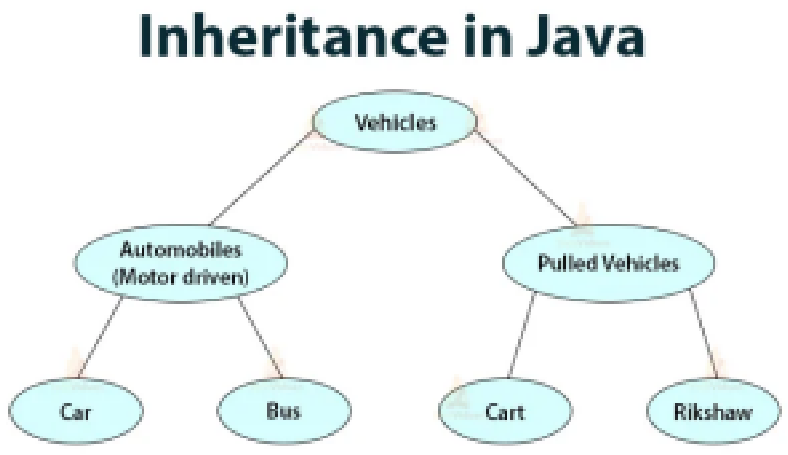
II. Inheritance

In the earlier sessions we have seen that in the real-world, inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.



In Object-Oriented Programming, Inheritance is an essential concept that enables the creation of new classes by inheriting properties and behaviors from an existing class. This promotes code reuse and follows the “is-a” relationship between classes. Inheritance allows a subclass (the derived class) to inherit from an existing superclass (the base class) to create a new class.

In Java, Inheritance is the way in which a class inherits fields and methods from another class. Inheritance can be of multiple levels. For example, if a class called Sedan extends a class called Car, and a class called Car extends a class called Vehicle, then the Sedan class inherits from the Vehicle class as well. In other words, the Sedan class inherits from both the Car and the Vehicle classes.



We can now even extend the “Car” class or other vehicles classes further by subclassing them.

Following key terminologies are commonly used:

- **Superclass/Base Class/Parent Class**: This is the class from which properties and behaviors are inherited.

- **Subclass/Derived Class/Child Class:** This is the new class that inherits properties and behaviors from the superclass. It can also add its own additional properties and behaviors.

Java uses the “extends” keyword while creating a class inheriting from another class. Here is a simple example:

Java
```java
class SuperclassName {
  //properties and behaviors of superclass
}

class SubclassName extends SuperclassName {
// inherited properties and behavior of superclass
// Additional properties and behaviors specific to the subclass
  ...
}
```

Benefits of Inheritance:

- **Code Reusability:** Inheritance allows you to reuse code from existing classes, reducing redundancy and improving maintainability.

- **Polymorphism:** Subclasses can be treated as instances of their superclass, allowing for polymorphic behavior.

- **Extensibility:** You can extend the functionality of a class without modifying the existing code.

III. Types of inheritance

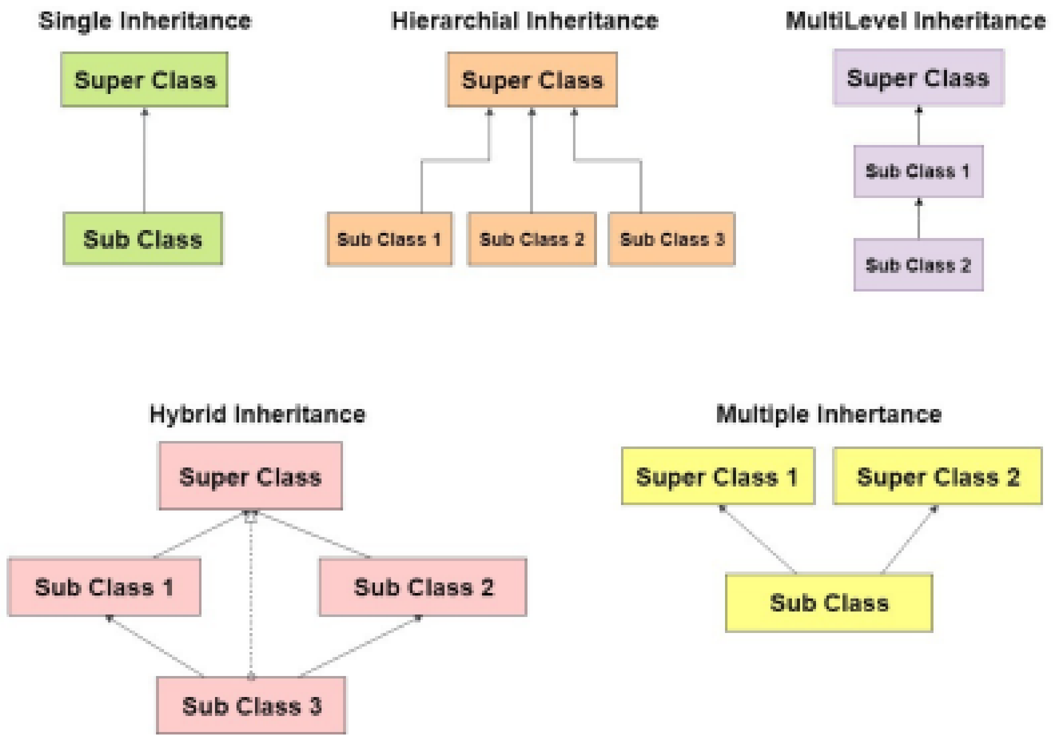
Java supports single inheritance while extending classes, which means a subclass can only inherit from one superclass.

Note: Java does not support multiple inheritance for classes but multiple inheritance can be achieved through the use of interfaces. Interface will be discussed in upcoming session.

Here are types of Inheritance supported by Java (Through classes and interfaces):

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

Following diagram will explain different types of inheritance:



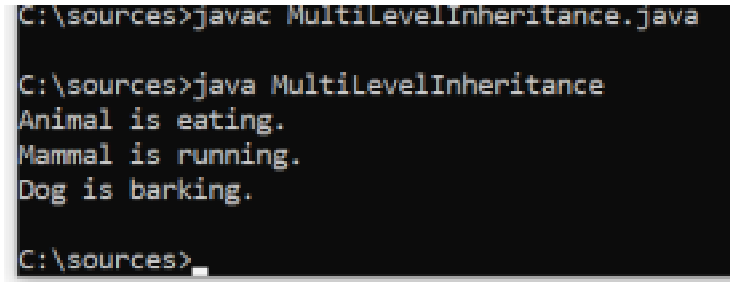
However, hybrid inheritance and multiple inheritance are not supported through class inheritance in Java. They are implemented through interfaces. It will be discussed after the introduction of interfaces.
Here is an example of multilevel inheritance in Java:

Java
```java
class Animal {
	void eat() {
```

```
                System.out.println("Animal is eating.");
        }
}
class Mammal extends Animal {
        void run() {
                System.out.println("Mammal is running.");
        }
}
class Dog extends Mammal {
        void bark() {
                System.out.println("Dog is barking.");
        }
}
public class MultiLevelInheritance {
        public static void main(String[] args) {
                Dog myDog = new Dog();
                myDog.eat(); // Inherited from Animal
                myDog.run(); // Inherited from Mammal
                myDog.bark(); // Dog's own method
        }
}
```

In the above example, we have an Animal class with the eat() method, a Mammal class with the run() method, and a Dog class with the bark() method. When an instance of the Dog class is created, it inherits all the methods from its superclasses.

Output:



```
C:\sources>javac MultiLevelInheritance.java

C:\sources>java MultiLevelInheritance
Animal is eating.
Mammal is running.
Dog is barking.

C:\sources>_
```

Access Control and Inheritance

In Java while creating a subclass from a base class following rules are applicable: ●

Public: Public members of a superclass are inherited by the subclass as public members. They can be accessed directly using the subclass instance. ●

Protected: Protected members of a superclass are inherited by the subclass as protected members. They can be accessed directly using the subclass instance, but only within the same package or by subclasses.

- **Private Members:** Private members of a superclass are not directly accessible in the subclass. They are hidden from the subclass and cannot be inherited or accessed.
- **Default (Package-private) Members:** Default members of a superclass (those without access modifiers) are inherited by the subclass within the same package. However, they are not inherited if the subclass is in a different package.

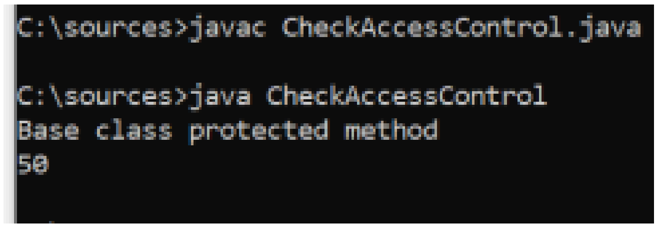
Java
```java
class Baseclass
{
       private int privateVariable = 30;
       protected int protectedVariable = 50;
       private void privateMethod()
       {
              System.out.println("Base class private method");
       }
       protected void protectedMethod()
       {
              System.out.println("Base class protected method");
       }
}
class Derivedclass extends Baseclass
{
}
public class CheckAccessControl {
       public static void main(String[] args)
       {
              // Creating an object of subclass.
              Derivedclass d = new Derivedclass();
              // We cannot access private members.
              // Private members of the base class are not available in the
subclass.
              // System.out.println("x = " +d.privateVariable);// Compilation
error
              // d.privateMethod(); // Compilation error
              // Accessing the protected members from the subclass.
              d.protectedMethod();
              System.out.println("y = " +d.protectedVariable);
       }
}
```

In the above example we have accessed the protected members of the base class. However, calling private members of the base class in derived classes will produce compilation errors.

Protected members can be available even after further subclass implementation. See the code below:

```java
class Baseclass
{
       private int privateVariable = 30;
       protected int protectedVariable = 50;
       private void privateMethod()
       {
              System.out.println("Base class private method");
       }
       protected void protectedMethod()
       {
              System.out.println("Base class protected method");
       }
}
class Derivedclass extends Baseclass
{
}
// Multilevel inheritance
class AnotherClass extends Baseclass {

}
public class CheckAccessControl {
       public static void main(String[] args)
       {
              // Protected class can be accessed even after further subclassing.
              AnotherClass an = new AnotherClass ();
              an.protectedMethod();
              System.out.println(an.protectedVariable);
       }
}
```

Output:



```
C:\sources>javac CheckAccessControl.java

C:\sources>java CheckAccessControl
Base class protected method
50
```

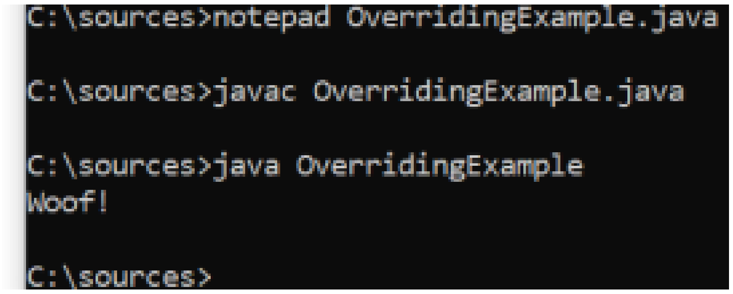
IV. Method overriding

Method overriding is a feature in Java that allows us to provide a different implementation of a method in the subclass that is already defined in its superclass. Method overriding is implemented through a method with the same name, same return type, and same parameter list as a method in its superclass.

Here's an example of method overriding in Java:

```java
class Animal {
       // Basic implementation defined in base class
   public void makeSound() {
   System.out.println("Animal is making sound");
 }
}
class Dog extends Animal {
  @Override
  public void makeSound() { // Method with same signature
    System.out.println("Woof!"); // Different implementation
 }
}
public class OverridingExample {

       public static void main(String[] args) {
              Dog dg = new Dog ();
              dg.makeSound();
       }
}
```

Output:



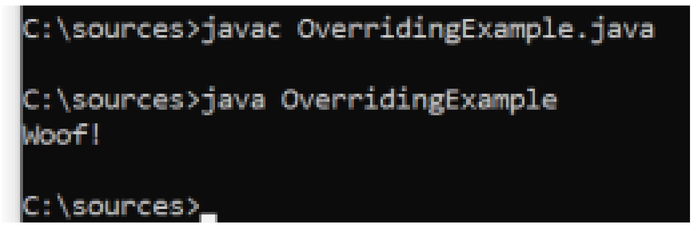
```
C:\sources>notepad OverridingExample.java

C:\sources>javac OverridingExample.java

C:\sources>java OverridingExample
Woof!

C:\sources>
```

In the above example the method makeSound() defined in base class Animal is overridden in the derived class Dog. The Animal class method makeSound(), simply prints a message "Animal is making sound". The Dog class extends the Animal class and overrides the makeSound() method to print "Woof!" instead. When we create a Dog object and call its makeSound method, the overridden method in the Dog class is executed instead of the method in the Animal class.

Java also allows us to create an instance of a derived class while using the reference of the base class. This behavior is called runtime polymorphism as actual implementation will be decided at runtime while creating the instance. It will be clear in the following example.

Java
```java
class Animal {
       // Basic implementation defined in base class
       public void makeSound() {
             System.out.println("Animal is making sound");
       }
}
class Dog extends Animal {
       @Override
       public void makeSound() { // Method with same signature
             System.out.println("Woof!"); // Different implementation
       }
}
public class OverridingExample {
       public static void main(String[] args) {
           Animal an = new Dog (); // An instance of Dog is assigned to
               // an instance of Animal
             an.makeSound();
       }
}
```

Output:



```
C:\sources>javac OverridingExample.java

C:\sources>java OverridingExample
Woof!

C:\sources>_
```

Access Control restriction in overriding methods in Java:

The following rules are inherited methods are enforced −

- Methods declared public in a superclass also must be public in all subclasses. ● Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them. In general, access modifiers can not be more restrictive in derived classes. The reason for this restriction is to ensure that a public or protected method in a base class is not blocked in the subclass. In Java all methods will execute the overridden method only. (This type of functions are called virtual functions or methods in object oriented programming. A virtual function or virtual method in OOP is a method or function that overrides the behavior of a function in an inherited class with the same signature to achieve the polymorphism.)

V. super()

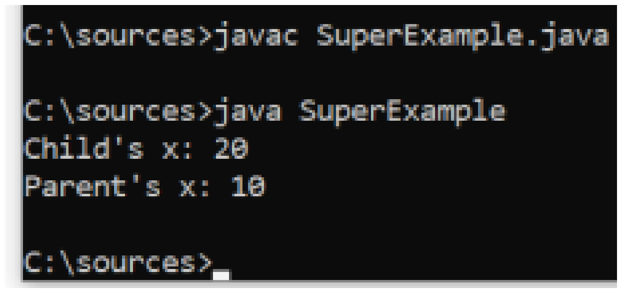
The super keyword is used in Java to refer to the immediate superclass of a subclass. The super keyword can also be used to access members of the superclass (field and methods) . The super keyword helps us to better control interactions between the subclass and the superclass. The super keyword is mainly used in the context of method overriding, constructors, accessing overridden methods, and fields.

Super keyword can be used to accessing fields of Superclass:

Java

```java
class Parent {
        int x = 10;
    }
    class Child extends Parent {
        int x = 20;
        void display() {
            System.out.println("Child's x: " + x);
            // Accessing parent attributes through super keyword
            System.out.println("Parent's x: " + super.x);
        }
    }
    public class SuperExample {
        public static void main(String[] args) {
            Child child = new Child();
            child.display();
        }
    }
```

Output:



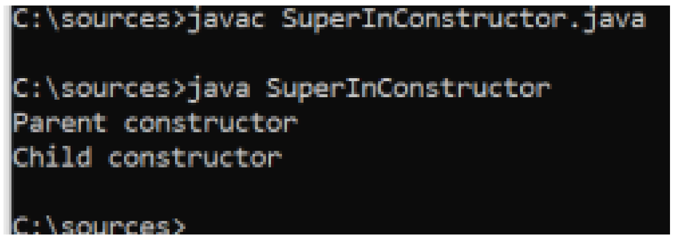
```
C:\sources>javac SuperExample.java

C:\sources>java SuperExample
Child's x: 20
Parent's x: 10

C:\sources>_
```

Super keyword can also be used to invoke the constructor of superclass:

Java

```java
class Parent {
      Parent() {
            System.out.println("Parent constructor");
      }
}
class Child extends Parent {
      Child() {
            super(); // Invokes parent constructor
            System.out.println("Child constructor");
      }
}
public class SuperInConstructor {
      public static void main(String[] args) {
            Child child = new Child();
      }
}
```

Output:



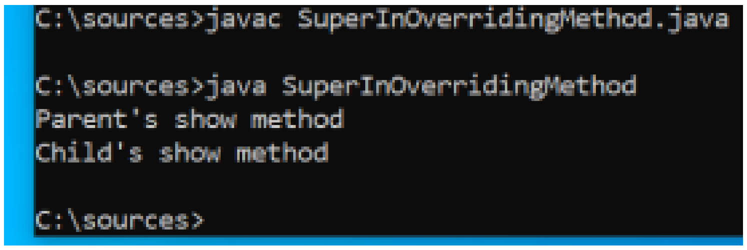
```
C:\sources>javac SuperInConstructor.java

C:\sources>java SuperInConstructor
Parent constructor
Child constructor

C:\sources>
```

Super keyword can also be used to call base class method from overridden method:

Java

```java
class Parent {

	void show() {

		System.out.println("Parent's show method");

	}

}

class Child extends Parent {

	@Override

	void show() {

		super.show(); // Calls parent's show method

		System.out.println("Child's show method");

	}

}

public class SuperInOverridingMethod {

	public static void main(String[] args) {

		Child child = new Child();

		child.show();

	}
}
```

The output screenshot:


```
C:\sources>javac SuperInOverridingMethod.java

C:\sources>java SuperInOverridingMethod
Parent's show method
Child's show method

C:\sources>
```

In summary, the super keyword in Java provides a way to interact with the superclass and its members from within a subclass. It's particularly useful for handling inheritance, method overriding, and constructor chaining.

After the 2 hr ILT, the student has to do a 1 hour live lab. Please refer to the Session 1 Lab doc in the LMS.