**PRACTICAL # 04**

**Understanding and Implementing Inheritance in Java**

**THEORY:**

Inheritance in Java allows a class to inherit from another class, also called extending a class. The class that extends is the subclass and the class being extended is the superclass.

Inheritance allows existing Code Reuse. It is an effective method to share code between classes that have some traits in common, yet allowing the classes to have some parts that are different.

For example, Vehicle class contains the fields and methods that all Vehicles need (e.g. the number plate, owner). Now two classes Car and Truck inherits from the class Vehicle.

A Car is a Vehicle. A Truck is a Vehicle. In practice, however, that is not how you determine which superclasses and subclasses application needs to have. This is typically determined by the business logic of the application. For instance, is it needed to refer to Car and Truck objects as Vehicle objects? Is it needed to process both Car and Truck objects uniformly? If it is then it makes sense to have a common Vehicle superclass for the two classes. If Car and Truck objects are not processed in the same way, they don't need common superclass, except in rare cases to share code between them (to avoid writing duplicate code).

**Class Hierarchies**

Superclasses and subclasses form an inheritance structure called class hierarchy. At the top the hierarchy is the superclasses and at the bottom of the hierarchy the subclasses. A class hierarchy may have multiple levels of superclasses and subclasses. A subclass may itself be a superclass of other subclasses.

**Java Inheritance Basics**

When a class inherits from a superclass, it inherits parts of the superclass methods and fields. The subclass can override the inherited methods. Fields cannot be overridden, but can be shadowed in subclasses.

When a subclass extends a superclass in Java, all protected and public fields and methods of the superclass are inherited by the subclass. protected and public fields can be called and referenced just like the methods declared directly in the subclass.

Fields and methods with default (package) access modifiers can be accessed by subclasses only if the subclass is located in the same package as the superclass. Private fields and methods of the superclass cannot be directly referenced, but referenced indirectly using the protected and public methods.

Constructors are not inherited by subclasses, but subclass constructor must call a superclass constructor.

**Java Singular Inheritance**

Java inheritance mechanism only allows a class to inherit from a single superclass. This is to avoid problems like the superclasses contain methods with the same names and parameters.

**Inheritance Syntax**

In Java inheritance is declared using the extends keyword. The extends keyword is used in the subclass definition. Here is Java inheritance example using the extends keyword:

public class Vehicle {

protected String numberPlate = null;

public void setNumberPlate(String licenseNumber) {

this.numberPlate = licenseNumber;

}

}

public class Car extends Vehicle {

int seatingCapacity = 0;

public int getSeatingCapacity() {

return this.seatingCapacity;

}

public void setSeatingCapacity(int seatingCapacity) {

this.seatingCapacity;

}

}

The Car class in this example extends or inherits the Vehicle class. The protected field numberPlate from Vehicle class is inherited by Car class and is now accessible inside a Car instance. The example shows accessing the numberPlate field in subclass inside the getNumberPlate() method. The getNumberPlate() method should be placed in the Vehicle class but the method is in the Car class just to show the protected field accessibility.

**Inheritance and Type Casting**

It is possible to reference a subclass as an instance of one of its superclasses. Using the class definitions from the previous example it is possible to reference an instance of the Car class as an instance of the Vehicle class. Because the Car class extends the Vehicle class, it is also a Vehicle.

Example:

Car car = new Car();

Vehicle vehicle = car;

This is possible because the Car class inherits from the Vehicle class.

It is possible to use an instance of subclass as if it is an instance of its superclass. The process of referencing an object of class as a different type than the class itself is called type casting. That is cast an object from one type to another.

**Upcasting and Downcasting**

Casting an object of a subclass to one of its superclasses is referred to as upcasting (from a subclass type to a superclass type).

It may also be possible to cast an object from a superclass type to a subclass type, but only if the object really is an instance of that subclass (or an instance of a subclass of that subclass). This is referred to as downcasting (from a superclass type to a subclass type).

**Example:**

Car car = new Car();

// upcast to Vehicle

Vehicle vehicle = car;

// downcast to car again

Car car2 = (Car) vehicle;

The following downcast example is, however, not valid. The Java compiler will accept it, but at runtime this code will throw a ClassCastException.

Truck truck = new Truck();

// upcast to Vehicle

Vehicle vehicle = truck;

// downcast to car again

Car car = (Car) vehicle;

The Truck object can be upcast to a Vehicle object, but it cannot be downcast to a Car object later resulting in a ClassCastException.

**Overriding Methods**

A subclass can override (redefine) methods of its superclass.

Example:

public class Vehicle {

String numberPlate = null;

public void setNumberPlate(String licenseNumber) {

this.numberPlate = licenseNumber;

}

}

public class Car extends Vehicle {

public void setNumberPlate(String license) {

this.numberPlate = license.toUpperCase();

}

}

After overriding, when the setNumberPlate() method is called on a Car object, the overridden method in the Car class is called. The method in the superclass is ignored. To override a method the method signature in the subclass must be the same as in the superclass. Otherwise a new method in the subclass will be created.

Private methods from superclass can't be overridden. If the superclass calls a private method internally from some other method, it will continue to call that method from the superclass, even if you create a private method in the subclass with the same signature.

**@override Annotation**

If you override a method in a subclass, and the method signature is changed in the superclass, the method in the subclass no longer overrides the superclass method. So it would be nice if the compiler can tell that the method being overridden no longer overrides a method in the superclass.

This is what the Java @override annotation is for. Place the Java @override annotation above the method that overrides a method in a superclass. Here is Java @override example:

public class Car extends Vehicle {

@override

public void setNumberPlate(String license) {

this.numberPlate = license.toUpperCase();

}

}

**Calling Superclass Methods**

If you override a method in a subclass, but still need to call the method defined in the superclass, you can do so using the super reference, like this:

public class Car extends Vehicle {

public void setNumberPlate(String license) {

super.setNumberPlate(license);

}

}

The method setNumberPlate() in the Car class, calls the setNumberPlate() method in the Vehicle class.

Superclass methods can be called from any method in a subclass. It does not have to be from the overridden method only.

**instanceof Instruction**

The instanceof keyword determines whether a given object is an instance of some class.

Example:

Car car = new Car();

boolean isCar = car instanceof Car;

isCar variable will contain the value true.

The instanceof can also be used determine if an object is a instance of a superclass of its class. Example below that checks if a Car object is an instance of Vehicle:

Car car = new Car();

boolean isVehicle = car instanceof Vehicle;

Assuming that the Car class extends (inherits from) the Vehicle class, the isVehicle variable will contain the value true after the code execution. A Car object is a Vehicle object because Car is a subclass of Vehicle. The instanceof instruction can be used to explore the inheritance hierarchy.

The variable type used with the instanceof instruction does not affect its outcome.

Car car = new Car();

Vehicle vehicle = car;

boolean isCar = vehicle instanceof Car;

Even though the vehicle variable is of type Vehicle, the object it ends up pointing to in this example is a Car object. Therefore the vehicle instanceof Car will evaluate to true.

Here is the same instanceof example, but using a Truck object instead of a Car object:

Truck truck = new Truck();

Vehicle vehicle = truck;

boolean isCar = vehicle instanceof Car;

The isCar will now contain value false. The Truck object is not a Car object.

**Fields and Inheritance**

Fields cannot be overridden in a subclass. If a field is defined in a subclass with the same name as in the superclass, the subclass field will hide (shadow) the field in the superclass. If the subclass accesses the field, it will access the field in the subclass. But if the subclass calls a method in the superclass, and that method accesses the field with the same name as in the subclass, the superclass field is now accessed.

Example below illustrates fields in subclasses shadowing superclass fields:

public class Vehicle {

String numberPlate = null;

public void setNumberPlate(String licensePlate) {

this.numberPlate = licensePlate;

}

public String getNumberPlate() {

return numberPlate;

}

}

public class Car extends Vehicle {

protected String numberPlate = null;

@Override

public void setNumberPlate(String license) {

super.setNumberPlate(license);

}

@Override

public String getNumberPlate() {

return super.getNumberPlate();

}

public void updateNumberPlate(String license){

this.numberPlate = license;

}

}

Notice how both classes have a numberPlate field.

The updateNumberPlate() method in the Car class accesses the numberPlate field directly. Thus, it accesses the numberPlate field of the Car class. Therefore, you will not get the same result if you call setNumberPlate() as when you call the updateLicense() method.

Car car = new Car();

car.setNumberPlate("123");

car.updateNumberPlate("abc");

System.out.println("license plate: " + car.getNumberPlate());

This Java code will print out the text 123.

The updateNumberPlate() method sets the number plate value on the numberPlate field in the Car class. The getNumberPlate() method, however, returns the value of the numberPlate field in the Vehicle class. Therefore, the value 123 which is set as value for the numberPlate field in the Vehicle class via the setNumberPlate() method, is what is printed out.

**Constructors and Inheritance**

Constructors of a superclass are not inherited by subclasses. Subclasses can still call the constructors in the superclass using the super() construct. In fact, a subclass constructor is required to call one of the constructors in the superclass as the very first action inside the constructor body. Here is how that looks:

public class Vehicle() {

public Vehicle() {

}

}

public class Car extends Vehicle{

public Car() {

super();

//perform other initialization here

}

}

This super() call executes the constructor in the Vehicle class.

Some subclass constructors do not seem to call the constructors in the superclass. The superclass may not even have a constructor. However, the subclass constructors have still called superclass constructors in those case.

If a class does not have any explicit constructor defined, the Java compiler inserts an implicit no-arg constructor. Therefore the following version of Vehicle is equivalent to the version shown just above:

public class Vehicle() {

}

If a constructor does not explicitly call a constructor in the superclass, the Java compiler inserts an implicit call to the no-arg constructor in the superclass. So the following version of the Car class is equivalent to the version shown before:

public class Car extends Vehicle{

public Car() {

}

}

In fact, this empty constructor can be leaved out and the Java compiler would insert it, and insert an implicit call to the no-arg constructor in the superclass. This is how the two classes would look then:

public class Vehicle() {

}

public class Car extends Vehicle{

}

If the Vehicle class did not have a no-arg constructor, but had another constructor which takes parameters, the Java compiler would complain. The Car class would then be required to declare a constructor, and inside that constructor call the constructor in the Vehicle class.

**Nested Classes and Inheritance**

The same Java inheritance rules apply to nested classes. Nested classes which are declared private are not inherited. Nested classes with the default (package) access modifier are only accessible to subclasses if the subclass is located in the same package as the superclass. Nested classes with the protected or public access modifier are always inherited by subclasses.

Nested class inheritance example:

class MyClass {

class MyNestedClass {

}

}

public class MySubclass extends MyClass {

public static void main(String[] args) {

MySubclass subclass = new MySubclass();

MyNestedClass nested = subclass.new MyNestedClass();

}

}

Notice it is possible to create an instance of the nested class MyNestedClass which is defined in the superclass (MyClass) through a reference to the subclass (MySubclass).

**Final Classes and Inheritance**

A final class cannot be extended. So no class can inherit the class below.

public final class MyClass {

}

**Abstract Classes and Inheritance**

An abstract class is a class that does not contain the full implementation of whatever the abstract class should do. Thus, it cannot be instantiated. In Java abstract classes are intended to be extended to create a full implementation. Thus, it is fully possible to extend an abstract class. The Java inheritance rules are the same for abstract classes as for non-abstract classes.

**Code Re-usability with Inheritance**

Inheritance allows reusablity of the existing code. Its one of the core concepts of the object-oriented programming. This feature allows to design a better code.

Inheritance preserves the existing feature of the original class and adds more functionality.

The example of the stack class done in the previous lab is a good candidate for inheritance. We can add more useful functions like how much is stack full now and clear all the items in the stack.

public class Stack1 extends Stack{

public int getTotalStackElements(){

//stackTop index at any point will be the total number of elements in the stack

return stackTop;

}

public void emptyStack(){

stackTop = 0; //this resets the stack

}

}

**ACTIVITIES**

**Activity 1**

Write a similar program that extends the class Queue and adds the functionality of getting queue size and clearing the queue.

**REVIEW QUESTIONS**

1. What are the major benefits of Inheritance?
2. What is upcasting and downcasting?
3. What is the importance of @override annotation?
4. What happens with constructors when a subclass object is created?