OBJECT-ORIENTED PROGRAMMING
LAB 10: NESTED CLASS, DESIGN PATTERN

I. Objective

After completing this lab tutorial, you can:

- Understand how to program with Nested Classes,
- Understand how to program with Design Patterns.

II. Nested Class

- Java allows defining a class within another class.

```
class Outer {
    ...
    class Nested {
        ...
    }
}
```

- The class **Outer** is the external enclosing class and the class **Nested** is the class defined within the class **Outer**.
- Nested classes are classified as static and non-static.
- Nested classes that are declared static are simply termed static nested classes whereas non-static nested classes are termed inner classes. To access the non-static nested class, create an object of the outer class, and then create an object of the inner class. Otherwise the static nested class without creating an object of the outer class.

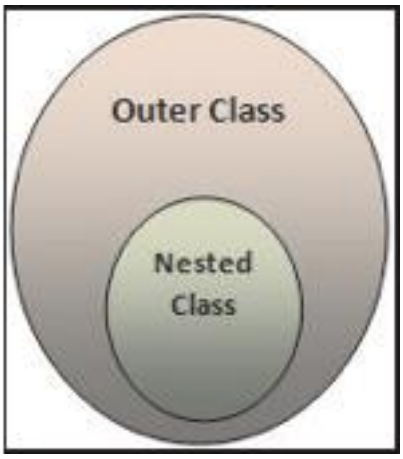
```
class Outer {
    ...
    static class StaticNested {
        ...
    }
    class Inner {
        ...
    }
}
```

- The class StaticNested is a nested class declared static whereas the non-static nested class, Inner, is declared without the keyword static.
- A nested class is a member of its enclosing class.

- Non-static nested classes or inner classes can access the enclosing class members even when it was declared private.
- Static nested classes cannot access any other member of the enclosing class.

1. Inner Class

1.1. Member Classes



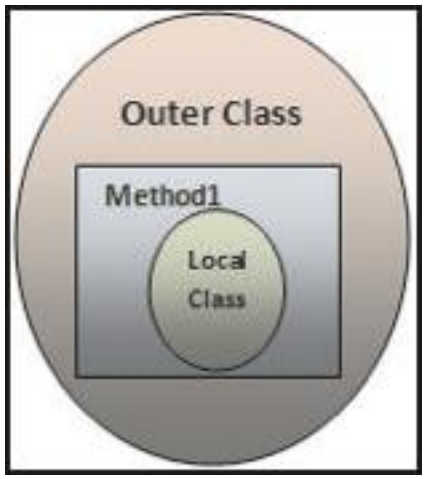
- A member class is a non-static inner class.
- It is declared as a member of the outer or enclosing class.
- The member class cannot have a static modifier since it is associated with instances of the outer class.
- An inner class can directly access all members that are, fields and methods of the outer class including the private ones.
- However, the outer class cannot access the inner class members directly even if they are declared public.
- This is because members of an inner class are declared within the scope of the inner class.
- An inner class can be declared as public, private, protected, abstract, or final.
- Instances of an inner class exist within an instance of the outer class.
- To instantiate an inner class, one must create an instance of the outer class.

```java
// Java program to demonstrate accessing
// an inner class

// outer class
class OuterClass {
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private int outer_private = 30;

    // inner class
    class InnerClass {
        void display() {
            // can access the static member of the outer class
            System.out.println("outer_x = " + outer_x);

            // can also access the non-static member of the outer class
            System.out.println("outer_y = " + outer_y);

            // can also access private members of the outer class
            System.out.println("outer_private = " + outer_private);

        }
    }
}

// Driver class
public class InnerClassDemo {
    public static void main(String[] args) {
        // accessing an inner class
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();

        innerObject.display();

    }
}
```

1.2. Local Class

- An inner class defined within a code block such as the body of a method, constructor, or initializer, is termed a local inner class.
- The scope of a local inner class is only within that particular block.

- Unlike an inner class, a local inner class is not a member of the outer class and therefore, it cannot have any access specifier.
- That is, it cannot use modifiers such as public, protected, private, or static.
- However, it can access all members of the outer class, as well as final variables, declared within the scope in which it is defined.

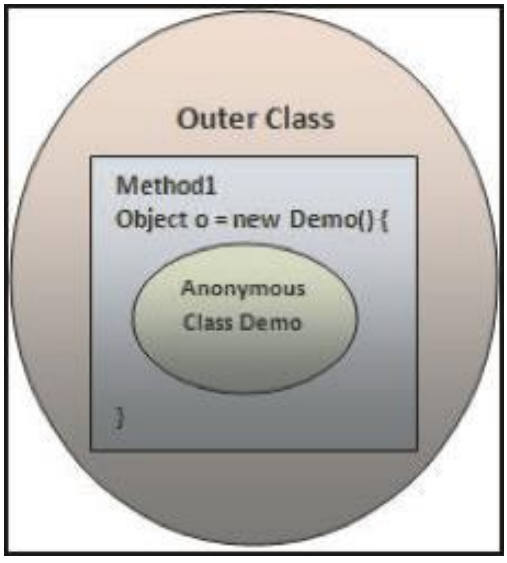


- The local inner class has the following features:
  - It is associated with an instance of the enclosing class.
  - It can access any members, including private members, of the enclosing class.
  - It can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition, provided that these are declared as final.

```
class OuterClass {
    int x = 10;

    public int methodA() {
        class InnerClass {
            int y = 5;
        }
        InnerClass myInner = new InnerClass();
        return x + myInner.y;
    }
}
public class MyMainClass {
    public static void main(String[] args) {
        OuterClass myOuter = new OuterClass();
        System.out.println(myOuter.methodA()); // Output: 15
    }
}
```

1.3. Anonymous Class

- An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.
- An anonymous class does not have a name associated, so it can be accessed only at the point where it is defined.
- It cannot use the *extends* and *implements* keywords nor can specify any access modifiers, such as public, private, protected, and static.
- It cannot define a *constructor, static fields, methods*, or *classes*.
- It cannot implement anonymous interfaces because an interface cannot be implemented without a name.
- Since an anonymous class does not have a name, it cannot have a named constructor but it can have an instance initializer.
- Rules for accessing an anonymous class are the same as that of the local inner class.
- Usually, an anonymous class is an implementation of its super class or interface and contains the implementation of the methods.
- Anonymous inner classes have a scope limited to the outer class.
- They can access the internal or private members and methods of the outer class.
- An anonymous class is useful for controlled access to the internal details of another class.
- Also, it is useful when a user wants only one instance of a special class.



```
interface Age {
    void getAge();
}

class AnonymousDemo {
    public static void main(String[] args) {

        // Myclass is a hidden inner class of the Age interface
        // whose name is not written but an object to it
        // is created.
        Age oj1 = new Age() {
            int x = 21;

            @Override
            public void getAge() {
                // printing age
                System.out.print("Age is " + x);
            }
        };
        oj1.getAge();
    }
}

// Output: Age is 21
```

2. Static Nested Class

- A static nested class is associated with the outer class just like variables and methods.
- A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but can access only through an object reference.
- A static nested class, by behavior, is a top-level class that has been nested in another top-level class for packaging convenience.
- Static nested classes are accessed using the fully qualified class name, that is, OuterClass.StaticNestedClass.
- A static nested class can have *public*, *protected*, *private*, *default*, or *package-private*, *final*, and *abstract* access specifiers.

```
// Java program to demonstrate accessing
// a static nested class

// outer class
class OuterClass {
    // static member
    static int outer_x = 10;
```

```
    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    static class StaticNestedClass {
        void display() {
            // can access the static member of the outer class
            System.out.println("outer_x = " + outer_x);

            // can access display private static members of the outer class
            System.out.println("outer_private = " + outer_private);

            // The following statement will give a compilation error
            // as static nested class cannot directly access non-static members
            // System.out.println("outer_y = " + outer_y);

        }
    }
}

// Driver class
public class StaticNestedClassDemo {
    public static void main(String[] args) {
        // accessing a static nested class
        OuterClass.StaticNestedClass nestedObject = new
        OuterClass.StaticNestedClass();

        nestedObject.display();

    }
}
```

III. Design Patterns

- A design pattern is a clearly defined solution to problems that occur frequently.
- Design patterns are based on the fundamental principles of object-oriented design.
- Following are the different types of design patterns:
    - Creational Patterns
    - Structural Patterns

- Behavioral Patterns

1. Singleton

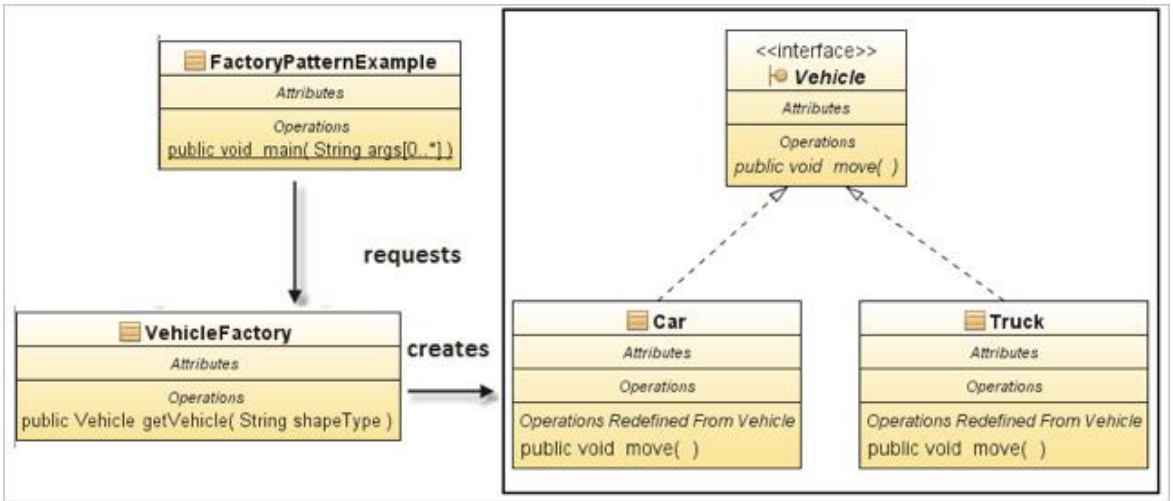
- A *singleton pattern* is a type of creational pattern.
- The *singleton design pattern* provides complete information on such class implementations.
- Consider the following when implementing the singleton design pattern:
  - The reference is finalized so that it does not reference a different instance.
  - The private modifier allows only same class access and restricts attempts to instantiate the singleton class.
  - The factory method provides greater flexibility. It is commonly used in singleton implementations.
  - The singleton class usually includes a private constructor that prevents a constructor to instantiate the singleton class.
  - To avoid using the factory method, a public variable can be used at the time of using a static reference.

```
class SingletonExample {
    private static SingletonExample singletonExample = null;

    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if (singletonExample == null) {
            singletonExample = new SingletonExample();
        }
        return singletonExample;
    }

    public void display() {
        System.out.println("Welcome to Singleton Design Pattern");
    }
}
```

```
public class SingletonTest {
    public static void main(String[] args) {
        SingletonExample singletonExample = SingletonExample.getInstance();
        singletonExample.display(); // Output: Welcome to Singleton Design Pattern
    }
}
```

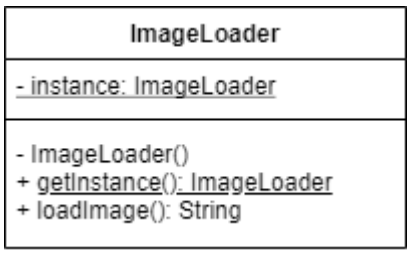
2. Factory Pattern

- It is one of the commonly used design patterns in Java.
- It belongs to the creational pattern category.
- This pattern does not perform direct constructor calls when invoking a method.
- The following figure shows the factory pattern diagram:



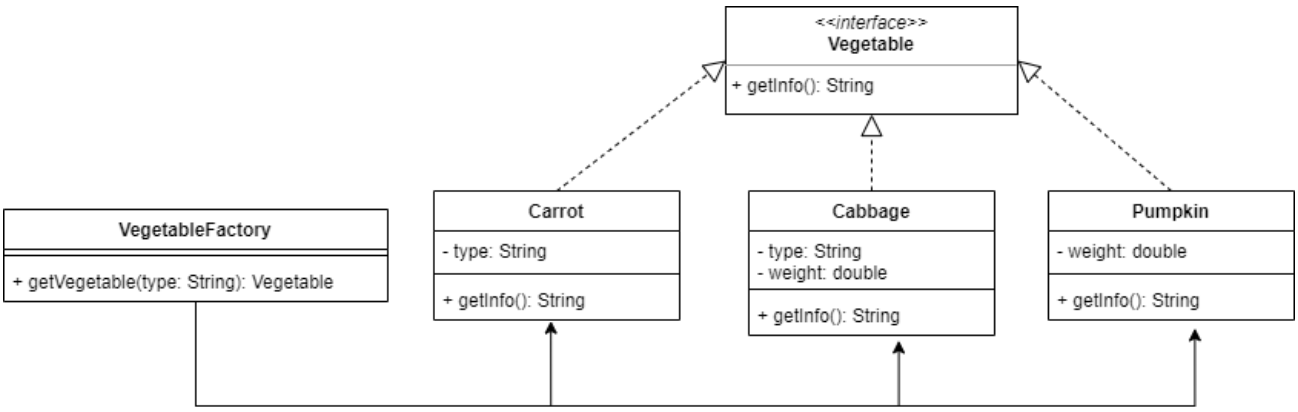
IV. Exercises

1. *Student* class contains the information *name*, *address*, *sex*, and *score*. The Student class has a nested class *StudentOperator*, that has two methods print() and type():
   - The print() method will print the following information:
     Student [" name "," address "," sex "," score "]
   - The type() method will return a student's rank with a data type of String.
     - If *the score* > 8 then return A,
     - If 5 <= *score* <= 8 then return B,
     - If the *score* < 5 then return C.
   - Write a main method to test your program.
2. We have an image loader application with **ImageLoader** class. This class responds to load the image. But only one instance of this **ImageLoader** class allow to exist in the program, let's define **ImageLoader** class with **Singleton Pattern** as the following class diagram.



A **loadImage()** method just needs to return the String "Loaded successfully." And write a main method to test your program.

3. Implement the program as the diagram below.



- getInfo(): this method returns the value of all properties of the class.

-- END--