OOP-II

Enum type

Inner Classes

Enum type

When to avoid constants

- public static final variables can be very useful, but there is a particular usage pattern you should avoid. Constants may provide a false sense of input validation or value range checking.
  - Consider a method that should receive only one of three possible values:

```
Computer comp = new Computer();
comp.setState(Computer.POWER_SUSPEND);
```

This is an `int` constant that equals `2`.

  - The following lines of code still compile:

```
Computer comp = new Computer();
comp.setState(42);
```

Without Enum

```
public class Computer {
public static final int POWER_OFF=0;
public static final int POWER_ON=1;
public static final int POWER_SUSPEND=2;

private int state;

public int getState() {
    return state;
}

public void setState(int state) {
    this.state = state;
}
```

```
public class ComputerTester {

public static void main(String[] args) {
Computer computer=new Computer();

computer.setState(Computer.POWER_ON);
System.out.println(computer.getState());

//Invalid state is accepted by the method
computer.setState(3);

}

}
```

Typesafe Enumerations

- A *Java Enum* is a special Java type used to define collections of constants. More precisely, a Java enum type is a special kind of Java class. An enum can contain constants, methods etc. Java enum were added in Java 5.
- Enums:
  - Provide a compile-time range check

```
public enum PowerState {
    OFF,
    ON,
    SUSPEND;
}
```

These are references to the only three PowerState objects that can exist.

- An enum can be used in the following way:

```
Computer comp = new Computer();
comp.setState(PowerState.SUSPEND);
```

This method takes a PowerState reference .

Note : Java enum extends java.lang.Enum class implicitly, so our enum types cannot extend another class.

Enum Usage

- Enum references can be statically imported.

```
import static com.example.PowerState.*;

public class Computer extends ElectronicDevice {
    private PowerState powerState = OFF;
    //...
}
```

PowerState.OFF

- Enums can be used as the expression in a switch statement.

```
public void showState(PowerState state) {
    switch(state) {
        case OFF:
        //...
    }
}
```

Typesafe Enumerations

```
public class Computer {
private PowerState powerState;


// getter and setter methods
public void showPowerState(PowerState state) {
        switch(state) {
        case OFF:  System.out.println("Computer is in sleep mode");
                break;
        ........
}
}}
```

```
public enum PowerState {
        OFF,ON,SUSPEND
}
```

```
public class EnumerationDemo{
  public static void main(String [] args){
        Computer computer = new Computer();
        //computer.setPowerState(1);    error
         computer.setPowerState(PowerState.ON);
         PowerState state= computer.getPowerState();
         computer.showPowerState(state);

         for(PowerState p : PowerState.values()){
         System.out.println(p+ ":"+ p.ordinal());
}
```

```
Computer is in active mode
OFF:0
ON:1
SUSPEND:2
```

Note: *values()* and *ordinal()* are built-in methods of Enumeration

Typesafe Enumerations

- *Enums can have fields, methods, and **private** constructors.*

```
public enum PowerStateNew {
    OFF("The power is off"),
    ON("The usage power is high"),
    SUSPEND("The power usage is low");

    private String description;
    private PowerStateNew(String d) {
        description = d;
    }
    public String getDescription() {
        return description;
    }
}
```

Call a `PowerState` constructor to initialize the `public static final OFF` reference.

The constructor should not be `public` or `protected`.

```
Computer computer= new Computer();
computer.setPowerStateNew(PowerStateNew.ON);
computer.getPowerStateNew().getDescription();
```

Benefits of enum

1) **enum is type-safe** you can not assign anything else other than predefined enum constants to an enum variable.

2) **enum** has its own name-space.

3) We **can use enum inside switch statement**

4) Adding new constants on **enum** is easy and we can add new constants without breaking existing code.

static import

What is static Import?

In order to access static members, it is necessary to qualify references with the class they came from. for example, one must say:

```
double r = Math.cos(Math.PI * theta);
```

You may want to avoid unnecessary use of static class members like math. and system. for this use static import. for example above code when changed using static import is changed to:

```
import static java.lang.System.out;
import static java.lang.Math.PI;
import static java.lang.Math.cos;
...
double r = cos(PI * theta);
...
```

Advantage: Enhances the **readability of the code.**

Nested Classes

Nested Classes

The Java programming language allows us to define a class within another class. Such a class is called as *nested class.*

Nested classes are divided into two categories:

1. static nested class
2. non-static nested class or inner class

Nested classes that are declared static are simply called as *static nested classes.*

Non-static nested classes are also called as *inner classes.*

Nested Classes

Static Nested class

```
<access-specifier> class OuterClassName {
   ...
  <access-specifier> static class StaticNestedClass {
    .......
}
}
```

Inner class

```
<access-specifier> class OuterClassName {
   ...
  <access-specifier> class InnerClassName {
    .......
    }
}
```

A nested class is a member of its enclosing class.

Non-static nested classes (inner classes) have access to the members (non-static as well as static) of the enclosing class(outer class), even if they are declared private.

Static nested classes do not have direct access to instance fields & instance methods of the outer class but they can directly access the static members of its outer class.

As a member of the Outer Class, a nested class can be declared private, public, protected, or *package private*.

(Recall that outer classes can only be declared public or *package private*.)

Nested Classes

Static Nested class

```
public class OuterClass {
  int a;
  static int b;
public  static class StaticNestedClass {
     int x;  // valid
     static int y; // valid
     System.out.println(b); // valid
     System.out.println(a); //  Error
}
}
```

Non-Static Nested class or inner class

```
public class OuterClass {
   int a;
   static int b;
public  class InnerClass   {
    int  x;  // valid
    static int y; // Error
    System.out.println(b); //valid
    System.out.println(a); //valid
    }
}
```

A static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Within a static nested class we can declare static as well as non-static members.

Static Nested Class

To create an object of a static nested class,

syntax:

```
OuterClass.StaticNestedClass object =  new OuterClass.StaticNestedClass();
```

Note: The static nested class has to have public access modifier

Note : Static nested classes are essentially like regular classes, except that their name is OuterClass.StaticNestedClass , hence provide some form of encapsulation that can not exactly be achieved with top-level classes.

Static nested class example

Demo Activity

Java Project : NestedClassesProject

```
public class OuterClass {
static String outerMessage="Welcome";

public static class StaticNestedClass{
 private  String innerMessage ="Hello";

 public String getInnerMessage(){
           return innerMessage + " " +outerMessage;
           }
 }

 public String getOuterMessage(){
           return new StaticNestedClass().innerMessage ;
     }
 }
```

The Enclosing class cannot directly access members of inner class. The following statement gives compilation error

```
return innerMessage;
```

```
public class NestedClassDemo{
 public static void main(String [] args){
        OuterClass outerClass = new OuterClass();
        System.out.println(outerClass.getOuterMessage());

        //creating instance of static nested class
        OuterClass.StaticNestedClass innerObject = new  OuterClass.StaticNestedClass();
        System.out.println(innerObject.getInnerMessage());
    }
}
```

Inner classes

Inner Classes

An instance of a outer class can exist on its own. By contrast, an instance of an inner class cannot be instantiated without being bound to its outer class.

Since an inner class is associated with an instance, it cannot define any static members itself.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

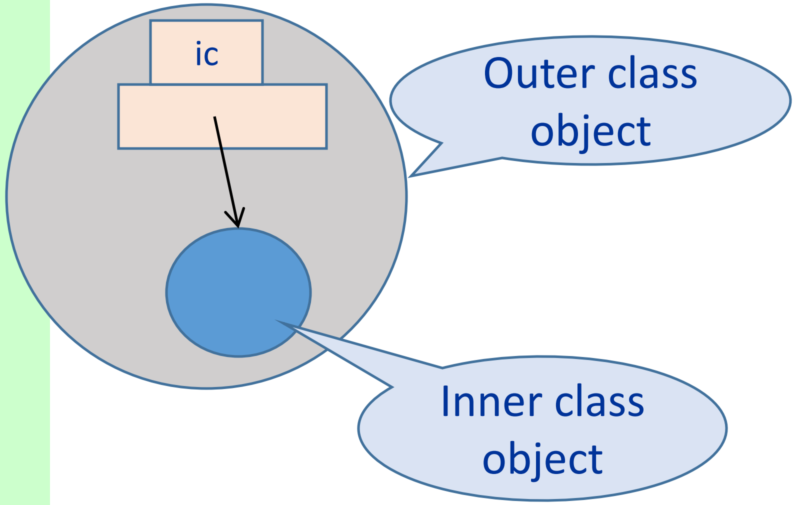
```
OuterClass.InnerClass innerObject =   outerClassObject.new InnerClass();
```

Inner Class Example

Demo Activity

```
public class OuterClassNew {
   private InnerClass ic;

   public OuterClassNew()    {
     ic = new InnerClass();
   }

   public void displayStrings()    {
     System.out.println(ic.getString() + ".");
     System.out.println(ic.getAnotherString() + ".");
   }

  public class InnerClass  {
   public String getString() {
      return "InnerClass: getString invoked";
   }

   public String getAnotherString() {
      return "InnerClass: getAnotherString invoked";
   }
  }
}
```

```
class InnerClassDemo{
public static void main(String[] args)   {
 OuterClassNew oc = new OuterClassNew();
 oc.displayStrings();
 OuterClassNew.InnerClass icObj = oc.new InnerClass();
 System.out.println(icObj.getString());
 }
}
```



Outer class methods cannot directly access the members of inner class. Solution: Access the inner class methods through inner class object reference

Note: If inner class is made private, then we cannot create instance of the inner class outside its enclosing class.

Local Inner classes and Anonymous Inner Classes

There are two more types of inner classes, i.e. local inner classes & anonymous inner classes.

- The local inner class are defined within a method.
- Anonymous inner classes are also defined with in a method *but have no name.*

Local Inner Classes

Syntax of the local inner class is as follows:

```
<access-specifier> class <ClassName> {
   …….

 <access-specifier> <return-type> <MethodName>(<arguments>){

      class <LocalInnerClassName>{
         ……
        }

      ……
   }
  ……
}
```

Anonymous Inner Class

Demo Activity

Syntax of the Anonymous Inner class is as follows:

```
(new SuperType(parameters) {
   inner class methods and data
}).method();
```

```
public class AnonymousInnerClass {

public static void main(String[] args) {
(new Object(){
public void display(String string){
System.out.println(string);
}
}).display("Welcome");
}
}
```

- Here, SuperType can be an interface, such as ActionListener; then, the anonymous inner class implements that interface.

- SuperType can be a class; then, the anonymous inner class extends that class.

- The anonymous inner class name will be the class name in which the anonymous inner class is created followed by $ symbol and 1.
- If the class name is *AnonymousInnerClass*, the anonymous inner class name will be *AnonymousInnerClass$1*

- In this example, we are performing the following:
    - Defining a class with a method in it
    - Creating instance of the class
    - Invoking the method through that instance

Anonymous Inner Class

Demo Activity

```
public class AnonymousClassTester {
private static String greet="Good Evening!";
public static void main(String[] args) {
 //Anonymous Inner Class
 (new Object(){
public void show(String message){
System.out.println( message+"Welcome To Anonymous Inner Classes");
}
}).show("Hi!");

 //Anonymous Inner Class
 //String greet="Good Evening!";  // Valid
 (new Object(){
 public void showGreet(String message){
System.out.println( message+"Welcome To Anonymous Inner Classes");
}
 }).showGreet(greet);

}
}
```

Anonymous Inner Class

Demo Activity

```
class A{
int a;
A(int a){
       this.a=a;
}
}

public class Anonym1 {
public static void main(String [] args){
(new A(10){
  public void showData(){
     System.out.println("a= "+ a);}
  }).showData();
}
}
```

Thank You!