

CENG 443
Introduction to Object-Oriented Programming
Languages and Systems

Nested Classes

Overview

- Sometimes we define a class whose purpose is only to “help out” another class.
- In that kind of situation, it would be nice if the helper class didn't stand on its own, but instead was more closely associated with the class itself.
 - Like being a part of the associated class

Nested classes

- Two categories: static and non-static.
- Nested classes that are declared static are called *static nested classes*.
- Non-static nested classes are called *inner classes*.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

Nested classes

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*.
- Outer classes can only be declared public or *package private*

Why Use Nested Classes?

- **It is a way of logically grouping classes that are only used in one place**
 - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**
 - Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

- A static nested class interacts with the instance members of its outer class (and other classes) **just like any other top-level class**.
- In fact, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static Nested Classes

- A class can be made **static** only if it is a nested (inner) class.
- Inner static class doesn't need reference of Outer class
- A static class cannot access non-static members of the Outer class

Outer class

```
public class StaticExample {  
    private static String str = "hello";
```

Inner class

```
    //Static class  
    static class MyNestedClass{  
        //non-static method  
        public void disp() {  
  
            /* If you make the str variable of outer class  
             * non-static then you will get compilation error  
             * because: a nested static class cannot access non-  
             * static members of the outer class.  
             */  
            System.out.println(str);  
        }  
    }  
}  
  
public static void main(String args[])  
{  
    /* To create instance of nested class we didn't need the outer  
     * class instance but for a regular nested class you would need  
     * to create an instance of outer class first  
     */  
    StaticExample.MyNestedClass obj = new StaticExample.MyNestedClass();  
    obj.disp();  
}
```

Inner Classes


- **Inner class** are defined inside the body of another class (known as **outer class**).
- These classes can have access modifier or even can be marked as abstract and final.
- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.
- Since an inner class is associated with an instance, it cannot define any static members itself.

Inner Class

- Inner class acts as a member of the enclosing class and can have any access modifiers: abstract, final, public, protected, private, static
- Inner class can access all members of the outer class including those marked private

```
//Top level class definition
class MyOuterClassDemo {
    private int myVar= 1;

    // inner class definition
    class MyInnerClassDemo {
        public void seeOuter () {
            System.out.println("Value of myVar is :" + myVar);
        }
    } // close inner class definition
} // close Top level class definition
```



Private
variables are
accessible

Instantiating an inner class

```
class MyOuterClassDemo {
    private int x= 1;
    public void innerInstance()
    {
        MyInnerClassDemo inner = new MyInnerClassDemo();
        inner. seeOuter();
    }
    public static void main(String args[]){
        MyOuterClassDemo obj = new MyOuterClassDemo();
        obj.innerInstance();
    }
    // inner class definition
    class MyInnerClassDemo {
        public void seeOuter () {
            System.out.println("Outer Value of x is :" + x);
        }
    } // close inner class definition
} // close Top level class definition
```

- To instantiate an instance of inner class, there should be a live instance of outer class.
- An inner class instance can be created only from an outer class instance.

Output:

```
Outer Value of x is :1
```

Instantiating an inner class from outside the outer class Instance Code

```
class MyOuterClassDemo {
    private int x= 1;
    public void innerInstance()
    {
        MyInnerClassDemo inner = new MyInnerClassDemo();
        inner. seeOuter();
    }
    public static void main(String args[]){
        MyOuterClassDemo.MyInnerClassDemo inner = new MyOuterClassDemo().new MyInnerClassDemo();
        inner. seeOuter();
    }
    // inner class definition
    class MyInnerClassDemo {
        public void seeOuter () {
            System.out.println("Outer Value of x is :" + x);
        }
    } // close inner class definition
} // close Top level class definition
```

Inner class types

- There are two special kinds of inner classes
 - local classes
 - defined in a *block*, which is a group of zero or more statements between balanced braces.
 - You typically find local classes defined in the body of a method
 - anonymous classes
 - are like local classes except that they do not have a name
 - Anonymous classes enable you to make your code more concise.
 - They enable you to declare and instantiate a class at the same time.
 - Use them if you need to use a local class only once

Local classes

- Local classes are similar to inner classes because they cannot define or declare any static members
- Local classes in static methods can only refer to static members of the enclosing class
- Local classes are non-static because they have access to instance members of the enclosing block
- A local class can have static members provided that they are constant variables

Example

```
//Top level class definition
class MyOuterClassDemo {
    String farewell = "Bye bye";

    public void sayGoodbyeInEnglish() {
        class EnglishGoodbye {
            public void sayGoodbye() {
                System.out.println(farewell);
            }
        }
        EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();
        myEnglishGoodbye.sayGoodbye();
    }
}

// close Top level class
```

Anonymous Inner Classes

- It is a type of inner class which
 - has no name
 - can be instantiated only once
 - is usually declared inside a method or a code block, a curly braces ending with semicolon
 - Is accessible only at the point where it is defined
 - does not have a constructor simply because it does not have a name
 - cannot be static

Example

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

btn.setOnAction specifies what happens when you select the button.

This method requires an object of type EventHandler<ActionEvent>. The EventHandler<ActionEvent> interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the btn.setOnAction method.