07_2 Nested Class

Object-Oriented Programming

Nested Class에 대해 강의하겠습니다.

Kind of Nested Class

- **Outer class** includes the nested class
- **Nested class** is defined within outer class (Static nested, Inner, Local Inner)

```
public class OuterClass {
    static class SNClass { } // Static nested class
    class InnerClass { }     // (non-static) Inner class
    void someMethod() {
        class LIClass { }    // Local Inner Class
        LIClass liObject = new LIClass(); // only used within the method
    }
}
public class OuterClassDemo {
    public static void main(String[] args) {
        OuterClass.SNClass snObject = new OuterClass.SNClass();
        OuterClass outObject = new OuterClass(); // create outer object first
        OuterClass.InnerClass inObject = outObject.new InnerClass();
        outObject.someMethod();
    }
}
```


페이지 2

Nested Class의 종류를 알아보기 위해
먼저 Outer class라는 용어를 사용하기로 합니다.
Outer class에는 그 안에 nested class를 가지고 있는
class를 지칭하는 것입니다.
Nested class는 static nested, inner,
local inner class로 나눌 수 있습니다.
먼저 OuterClass 안에 static nested class라는 것이 있는데
static 이므로 OuterClass의 모든 object들이
공유하고 있는 nested class입니다.
다음에 Inner class가 있는데
non-static으로 OuterClass 안에 포함된 class를 말합니다.
마지막으로 어떤 method 안에서 정의된
Local inner class가 있습니다.
이 local inner class는 오로지 이 method 내에서만 사용 가능합니다.
이 세가지 nested class들의 type과 생성하는 방법이 모두 다릅니다.
우선 static nested class인 SNClass는 OuterClass.SNClass type이며
new OuterClass.SNClass() 로 간단히 생성 가능합니다.
이것은 static nested class가 OuterClass의 모든 object들이
공통으로 가지고 있는 class이며, 따라서 object도 유일하게 존재하기 때문입니다.
한편 non-static inner class는 OuterClass.InnerClass type인데
이 inner class의 object를 생성하기 위해서는
우선 OuterClass object인 outObject를 먼저 생성해야 합니다.
이렇게 outObject를 생성한 후
inner class의 object인 inObject는
outObject.new InnerClass() 로 생성 가능합니다.
마지막으로 local inner class는
OuterClass object인 outObject에서
someMethod를 outObject.someMethod() 로 call하면서
생성할 수 있습니다.

Example) Nested Class (1/3)

```java
public class OuterClass {

    static class SNClass { // Static nested class
        void display() {
            System.out.println("Inside static nested class");
        }
    }

    class InnerClass { // Inner class (member inner class)
        void display() {
            System.out.println("Inside inner class");
        }
    }
```


페이지 3

이제 nested class를 좀 더 구체적으로 사용하는 예를 보겠습니다.
OuterClass 가 있고
그 안에 static nested class인 SNClass가 있습니다.
SNClass 에는 display() method가 있는데
"Inside static nested class" 라는 메시지를 출력합니다.
그 아래에는 non-static인 InnerClass가 정의되어 있습니다.
이 안에도 display() method가 있는데
"Inside inner class" 라는 메시지를 출력합니다.

Example) Nested Class (2/3)

```
public class OuterClass {

    . . .

    void myMethod() {
        class LIClass { // Local Inner Class
            void display() {
                System.out.println("Inside local inner class");
            }
        }
        LIClass liObject = new LIClass(); // should be used within the method
        liObject.display();
    }
}
```


페이지 4

그 다음에 myMethod() method 안에
local inner class인 LIClass가 있습니다.
이 안에 포함된 display() method에서는
"Inside local inner class" 라는 메시지를 출력합니다.
이 local inner class를 생성하는 것은
오직 myMethod() method 내에서만 가능하기 때문에
여기에 LIClass object인 liObject를 생성하고
liObject.display() 를 call하는 것 까지의 code가
myMethod() 안에 포함되어 있습니다.

Example) Nested Class (3/3)

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

        // directly create OuterClass.SNClass
        OuterClass.SNClass snObject = new OuterClass.SNClass();
        snObject.display(); // OUTPUT: “Inside static nested class”

        OuterClass outObject = new OuterClass(); // create outer object first

        // using outObject.new to create innerClass's object
        OuterClass.InnerClass inObject = outObject.new InnerClass();
        inObject.display(); // OUTPUT: “Inside inner class”

        outObject.myMethod(); // OUTPUT: “Inside local inner class”
    }
}
```


페이지 5

한편 OuterClassDemo class의 main method 안에서
static nested class의 object인 snObject를 생성합니다.
이 snObject는 OuterClass.SNClass type이며
new OuterClass.SNClass() 로 생성이 가능합니다.
생성 후에 snObject.display() 를 call하여
“Inside static nested class” 라는 메시지를 출력했습니다.
그 아래에는 OuterClass의 object인 outObject를
new OuterClass() 로 생성하였습니다.
non-static class object인 inObject는 OuterClass.InnerClass type이고
outObject.new InnerClass() 로 생성할 수 있습니다.
inObject.display() 를 call하면 “Inside inner class” 를 출력합니다.
마지막으로 local inner class를 생성하기 위해
outObject.myMethod() 를 call하면
그 안에서 local inner class인 LIObject를 생성해 주고
“Inside local inner class” 를 출력합니다.

Example) (Non-static) Inner Class

```
public class AClass {
    public class BClass {
        public class CClass {  }
    }
    public static void main(String[] args) {
        AClass aObject = new AClass();
        AClass.BClass bObject = aObject.new BClass();
        AClass.BClass.CClass cObject = bObject.new CClass();
    }
}
```


페이지 6

이 slide에 있는 code는
non-static inner class를 계속 nested로 생성하는 예를 보여주고 있습니다.
가장 바깥쪽의 outer class는 AClass 이고
그 안에 BClass가 정의되어 있고
또 BClass 안에 CClass가 정의되어 있습니다.
이제 main method에서
AClass aObject = new AClass(); 로
가장 outer class인 aObject 를 생성하였습니다.
그 아래에는 AClass.BClass bObject = aObject.new BClass() 로
inner class를 생성하였습니다.
또 BClass의 inner class인 CClass object를 생성하는 것은
AClass.BClass.CClass cObject = bObject.new CClass()
와 같이 할 수 있습니다.

Rules for Nested Class

- **Name of an inner class**
  - cannot be reused inside the outer class
- **Private inner class**
  - cannot be accessed by name outside the the outer class
  - should be accessed through the public (package) method
- **Private variables and methods** of inner and outer classes
  - can access of each other by name

페이지 7

Nested class들을 위한 몇가지 중요한 rule들을 살펴 보겠습니다.
먼저 inner class의 name은 outer class 안에서 다시 사용될 수 없습니다.
inner class가 private 이라면
outer class의 외부에서 direct하게 inner class를 이름으로 access할 수 없습니다.
private inner class를 access하려면 public이나 package 권한의 method인
accessor method를 통해 접근할 수 있습니다.
그러나 inner와 outer class의 private variable과 private method들은
제한없이 서로 다른쪽에서 access가 가능합니다.

Example) Private Inner Class

```
class OuterClass {
    private class InnerClass { }
    void createInnerObject() {
        InnerClass inner = new InnerClass();
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.createInnerObject();  // OK
        // OuterClass.InnerClass inner = outer.new InnerClass();
        // compile error!! cannot access from outside directly
    }
}
```


페이지 8

Private Inner Class의 사용 예를 보겠습니다.
OuterClass 안에 InnerClass가 private 으로 정의되어 있습니다.
이럴때 OuterClass의 외부에서 InnerClass object를 만들고 싶을때
사용할 수 있도록, createInnerObject를 package access로 정의해 놓았습니다.
OuterClass의 외부인 Main class 내에서
먼저 OuterClass object인 outer 를 생성하고
outer.createInnerObject() 를 call하여
private inner class object를 생성하도록 했습니다.
그러나 OuterClass.InnerClass inner = outer.new InnerClass() 와 같이
inner class를 직접 생성하려 한다면
compile error가 일어나게 됩니다.
왜냐하면 InnerClass 가 OuterClass 내에 private으로 정의되어 있기 때문입니다.

Example) Outer, Inner Private Members

```
class OuterClass {
    private String outerPrivateVar = "Outer Private Variable";
    class InnerClass {
        private String innerPrivateVar = "Inner Private Variable";
        void accessOuterClass() {
            System.out.println(outerPrivateVar); // Outer's private access, OK
        }
    }
    void accessInnerClass() {
        InnerClass inner = new InnerClass();
        System.out.println(inner.innerPrivateVar);// Inner's private access, OK
    }
}
```


페이지 9

먼저 OuterClass의 private String variable인
outerPrivateVar에 "Outer Private Variable" 이라는
String을 assign 합니다.
내부에 정의된 InnerClass 안에
역시 private인 innerPrivateVar를 declare하고 초기화 합니다.
accessOuterClass() method에서는
InnerClass 안에서 OuterClass의 private variable을
직접 access할 수 있음을 보여줍니다.
한편 OuterClass의 accessInnerClass() 에서
InnerClass object를 생성하고
그것의 private instance variable을
직접 access할 수 있음을 보여주고 있습니다.

Anonymous Class

- Only need to implement an interface once
  - implemented class is **not reused** elsewhere
- Easier to understand
  - class implementation is just near the variable
- Implement callback method
  - used in GUI applications such as button click

페이지 10

마지막으로 Anonymous Class에 대해 알아보겠습니다.
Anonymous Class는 어떤 interface를
딱 한번만 implement해야 하는 경우에 사용될 수 있으며
이 경우에 impelement된 class는
다른 어느 곳에서도 재 사용되지 않습니다.
그러니까 우리 생활에서 어떤 순간 어떤 장소에 꼭 필요한 도구이지만
그 때 말고는 다시 어느 경우에도 사용되지 않는 도구라면
수공품으로 만들어서 한번만 사용할 수 있도록 하면되고
설계도를 공장으로 보내 여러 오브젝트를 만들 틀까지 준비할 필요는
없는 경우라고 비유할 수 있습니다.
Anonymous class는 class의 implementation이
variable의 옆에 바로 놓여져 있으므로
어떻게 보면 이해하기 쉽다고도 할 수 있습니다.
Anonymous class는 callback method의 구현에 많이 사용되는데,
특히 Graphical User Interface (GUI) application에서
button click과 같은 event가 발생했을 때 실행되는
callback method를 구현하는데서 볼 수 있습니다.

Example) Anonymous Class

```
interface Computer {
    void compute();
}
public class AnonymousClassDemo {
    public static void main(String[] argc) {
        Computer computer1 = new Computer() { // anonymous class
                                public void compute() {
                                  System.out.println("This is the computer1");
                                }
                             };
        Computer computer2 = new Computer() {
                                public void compute() {
                                  System.out.println("This is the computer2");
                                }
                             };
        computer1.compute(); // OUTPUT: “This is the computer1”
        computer2.compute(); // OUTPUT: “This is the computer2”
    }
}
```


페이지 11

Anonymous class의 예제 프로그램입니다.
먼저 interface Computer에는 compute() method가 존재합니다.
AnonymousClassDemo class의 main에서
Computer interface를 implement하여
computer1이라는 object를 생성하였습니다.
그런데 new 다음에 어딘가에 정의되어 있는 class 이름을 쓴 것이 아니고
anonymous class를 사용하여 “This is the computer1” 이라는
String을 output하는 compute method를 implement한
class object를 직접 생성하였습니다.
이와 같이 class definition을 따로 두지 않고
class object 생성할 때 한번만 사용하도록 하는 방법이
anonymous class 입니다.
그 아래의 computer2 object도
“This is the computer2” 라는 String을 print하는
compute method를 implement한
anonymous class를 사용하여
그 object를 생성하였습니다.
이제 computer1.compute() 를 call 하면
“This is the computer1” 이 print되고
computer2.compute() 를 call하면
“This is the computer2”가 print됩니다.