



# Object Orientation Fourth Story

Bok, Jong Soon  
[javaexpert@nate.com](mailto:javaexpert@nate.com)

<https://github.com/swacademy/Core-Java>

## **abstract Methods**

- Java allows you to specify that a superclass declares a method that does not supply an implementation.
- The implementation of this method is supplied by the subclasses.
- This is called an **abstract** method.

## abstract Classes

- A class that declares the existence of methods but not the implementation is called **abstract** class.
- Any class with one or more abstract methods is called an **abstract** class.
- You can declare a class as abstract by marking it with the **abstract** keyword.
- An **abstract** class can contain member variables and non-abstract methods.
- Use abstract as a modifier on all classes that should never be instantiated.
  - Mammal
  - Vehicle

# abstract Classes (Cont.)

## ✓ 추상이란?

- 여러 가지 사물이나 개념으로부터 공통이 되는 특성을 파악하는 것을 말합니다.

## ✓ 추상 클래스

- abstract 키워드를 이용하여 구현되지 않은 추상 메소드를 기술할 수 있습니다.
- 추상 메소드를 하나 이상 가지고 있는 클래스는 추상클래스가 되어야 합니다.
- 자식클래스로 구현을 미루고자 할 때 사용하며, 프로그램의 확장성을 위해 많이 사용하는 설계 방법입니다.

## ✓ 추상 클래스의 상속

- 추상 클래스를 상속받은 클래스는 추상 메소드를 반드시 구현해야 합니다.
- 단, 상속받은 클래스가 추상클래스가 되는 경우 구현하지 않아도 됩니다.

## ✓ 추상 클래스의 객체 생성

- abstract 클래스는 객체를 생성할 수 없습니다.
- 상속을 통하여 추상 메소드를 구현하여 객체를 생성할 수 있습니다.

```
public abstract class AstDog {  
  
    public abstract void bark();  
  
}  
  
public class SabsalDog extends AstDog {  
    public void bark(){  
        System.out.print("왈왈");  
    }  
}
```

AstDog.java

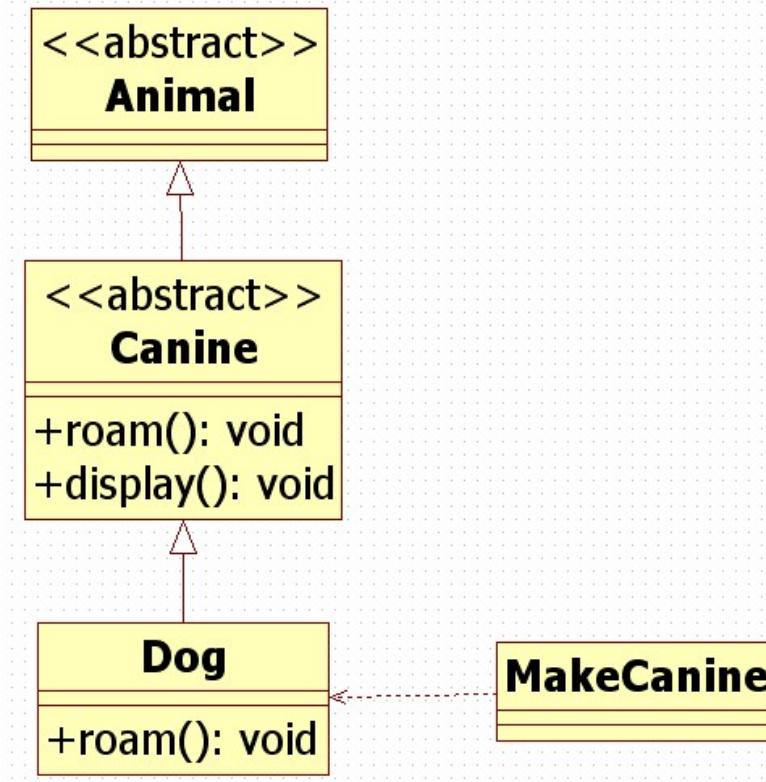
## abstract Classes (Cont.)

```
1 abstract class Animal{}  
2 abstract class Canine extends Animal{  
3     public abstract void roam();  
4     public void display(){  
5         System.out.println("I'm method");  
6     }  
7 }  
8 public class MakeCanine {  
9     public void go(){  
10        Canine c;  
11        c = new Canine();  
12        c.roam();  
13    }  
14 }
```

```
----- Java Compiler -----  
MakeCanine.java:11: Canine is abstract; cannot be instantiated  
        c = new Canine();  
                           ^  
1 error
```

# abstract Classes (Cont.)

```
1 abstract class Animal{}
2 abstract class Canine extends Animal{
3     public abstract void roam();
4     public void display(){
5         System.out.println("I'm method");
6     }
7 }
8 class Dog extends Canine{
9     public void roam(){
10        System.out.println("I'm a Dog's Method");
11    }
12 };
13 public class MakeCanine {
14     public void go(){
15         Canine c;
16         c = new Dog();
17         c.roam();
18     }
19 }
```



## interfaces

- An **interface** is a variation on the idea of an abstract class.
- In an **interface**, all the methods are **abstract**.
- In an **interface**, all variables are constant.
- You can simulate multiple inheritance by implementing such interfaces.

## interfaces (Cont.)

- All override method's access modified is **public**.
- You cannot use **final**, **abstract** together.
- You can implement **interface** using **implements** keyword.
- You can be initialized using polymorphism instead of using **new**.
- You can inherit multiple inheritance interfaces using **extends** keyword.
- Interface may use array.

# interfaces (Cont.)

## ✓ 인터페이스

- interface 키워드는 abstract 클래스의 개념에서 한 걸음 더 나아가 메소드 자체를 정의할 수 없게 합니다. 즉, 모든 메소드가 추상 메소드가 됩니다.
- 인터페이스는 설계와 구현을 완전히 분리합니다.
- 인터페이스는 원하는 만큼 조합하여 사용할 수 있지만, 일반 클래스나 추상 클래스는 다중 상속을 할 수 없습니다.

```
public interface IfDog {  
  
    void bark();  
  
}  
  
public class Dog implements IfDog {  
  
    @Override  
    public void bark(){  
        System.out.print("왈왈");  
    }  
}
```

IfDog.java

# interfaces (Cont.)

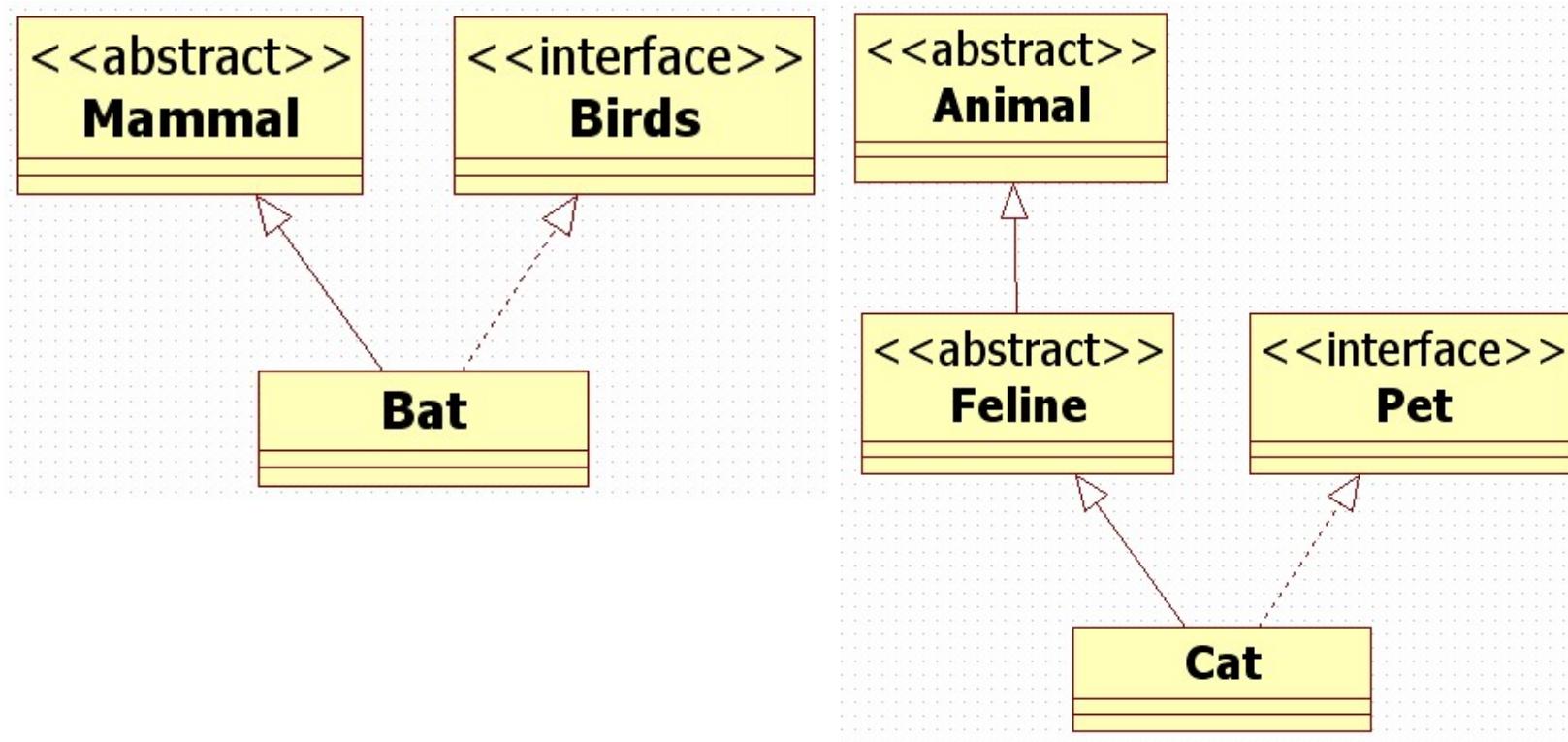
## ✓ 추상 클래스와 인터페이스를 사용하는 이유입니다.

- 설계 시 선언해 두면 개발할 때는 기능 구현에만 집중할 수 있습니다.
- 개발자에 따른 메소드명과 매개변수 선언의 차이를 줄일 수 있습니다.
- 공통적인 인터페이스와 추상클래스를 선언해 두면, 설계과 구현을 분리할 수 있습니다.

구분	클래스	추상클래스	인터페이스
예약어	class	abstract class	interface
추상 메소드 포함 여부	포함 불가	포함 가능	포함 가능 (필수)
메소드 포함 여부	포함 가능	포함 가능	포함 불가
Static 메소드 선언 여부	선언 가능	선언 가능	선언 불가
Final 메소드 선언 여부	선언 가능	선언 가능	선언 불가
상속 가능 여부	상속 가능	상속 가능	상속 불가
구현 가능 여부	구현 불가	구현 불가	구현 가능

※ 클래스, 추상클래스, 인터페이스 비교

# interfaces (Cont.)



# Polymorphism using interface

```
2 public interface Petable {  
3     void pet();  
4 }
```

```
2 public class Cat implements Petable {  
3     private String name;  
4     public Cat(String name){  
5         this.name = name;  
6     }  
7     @Override  
8     public void pet() {  
9         System.out.println("Cat " + this.name + " is very pretty.");  
10    }  
11 }
```

```
2 public class Dog implements Petable {  
3     private String name;  
4     public Dog(String name){  
5         this.name = name;  
6     }  
7     @Override  
8     public void pet() {  
9         System.out.println("Dog " + this.name + " is so cute.");  
10    }  
11 }
```

## Polymorphism using interface (Cont.)

```
2 public class PolymorphismDemo {  
3  
4     public static void main(String[] args) {  
5         PolymorphismDemo pd = new PolymorphismDemo();  
6         Petable dog = pd.create("Dog", "Duncan");  
7         dog.pet();  
8         Petable cat = pd.create("Cat", "Candy");  
9         cat.pet();  
10    }  
11    public Petable create(String kind, String name){  
12        if(kind.equals("Dog")){  
13            return new Dog(name);  
14        }else{  
15            return new Cat(name);  
16        }  
17    }  
18 }
```

Dog Duncan is so cute.  
Cat Candy is very pretty.

## Polymorphism using interface (Cont.)

```
2 public interface Vehicle {  
3     void drive();  
4 }
```

```
2 public class Car {  
3     public void carDrive(Vehicle v) {  
4         v.drive();  
5     }  
6 }
```

```
2 public class Matiz implements Vehicle {  
3     @Override  
4     public void drive() {  
5         System.out.println("Matiz is driving...");  
6     }  
7 }
```

```
2 public class Sonata implements Vehicle {  
3     @Override  
4     public void drive() {  
5         System.out.println("Sonata is driving...");  
6     }  
7 }
```

## Polymorphism using interface (Cont.)

```
2 public class PolymorphismDemo1 {  
3     public static void main(String[] args) {  
4         Car car = new Car();  
5         car.carDrive(new Matiz());  
6         car.carDrive(new Sonata());  
7     }  
8 }
```

Matiz is driving...  
Sonata is driving...

# Object Clone with interface

- ✓ Object 클래스에는 인스턴스 복사를 위한 기능이 있습니다.

- Object 클래스의 clone() 메소드

```
protected Object clone() throws CloneNotSupportedException
```

- 이 메소드가 호출되면, 이 메소드가 호출된 인스턴스의 복사본이 생성되고 복사본의 참조 값이 반환됩니다.



## Object Clone with interface (Cont.)

- ✓ 객체 복제 기능을 제공하는 클래스를 작성하는 방법을 알아봅니다.
  - Cloneable 인터페이스를 구현합니다.
  - clone() 메소드를 재정의한다. 이때, 접근 제한자를 public 으로 변경합니다.
    - 다른 클래스에서 clone()을 호출할 수 있도록 하기 위해서입니다.

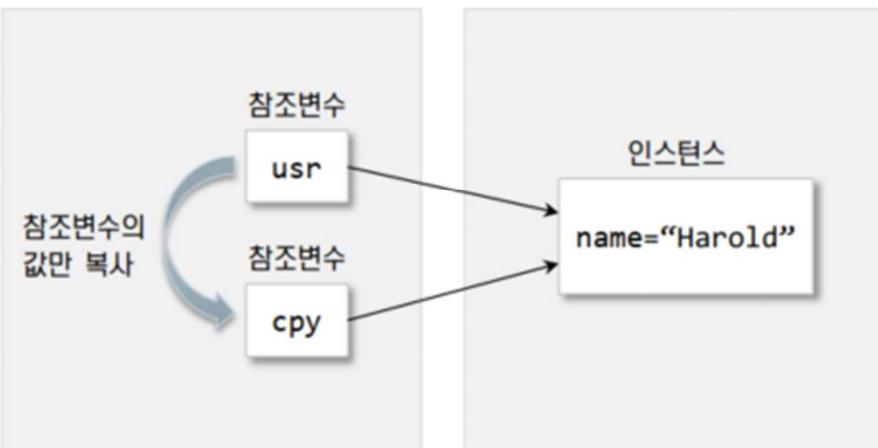
```
public class User implements Cloneable {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

ItDog.java

## Object Clone with interface (Cont.)

- ✓ 참조변수 복사와 객체 복사의 차이점입니다.
- ✓ 참조변수를 대입연산자를 통해서 복사하게 되면 참조하는 주소값만 복사되어서 동일한 인스턴스를 가리키게 됩니다.
- ✓ clone() 메소드로 객체를 복사하면 인스턴스의 복사본이 생성되어 각각 다른 인스턴스를 가리키게 됩니다.

```
User usr = new User();
usr.setName("Harold");
User cpy = usr;
```



Stack 영역

Heap 영역

```
User usr = new User();
usr.setName("Harold");
User cpy = (User) usr.clone();
```



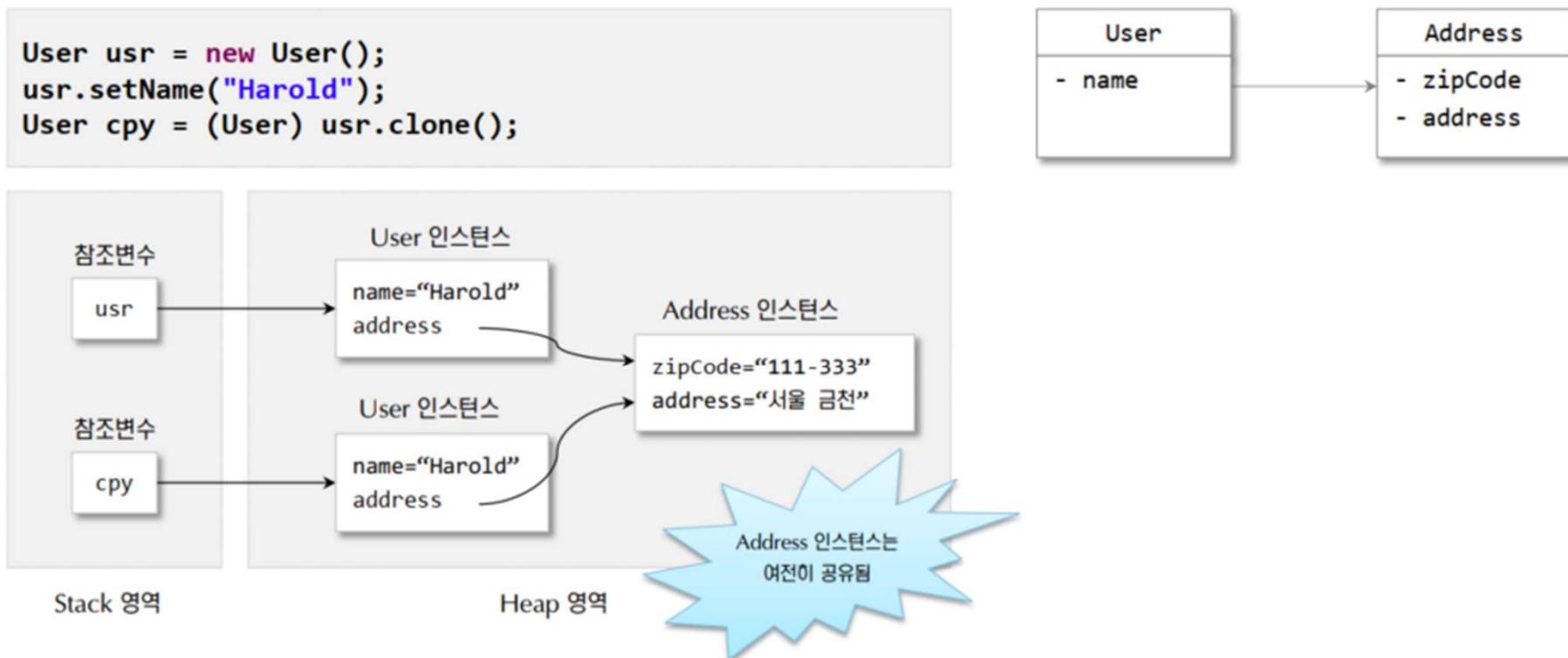
Stack 영역

Heap 영역

# Object Clone with interface (Cont.)

## ✓ Shallow Copy의 문제점

- 객체 필드가 객체를 가지는 경우
  - 사용자 객체가 주소 객체를 가지고 있는 경우에 주소 객체는 여전히 같은 인스턴스를 참조합니다.

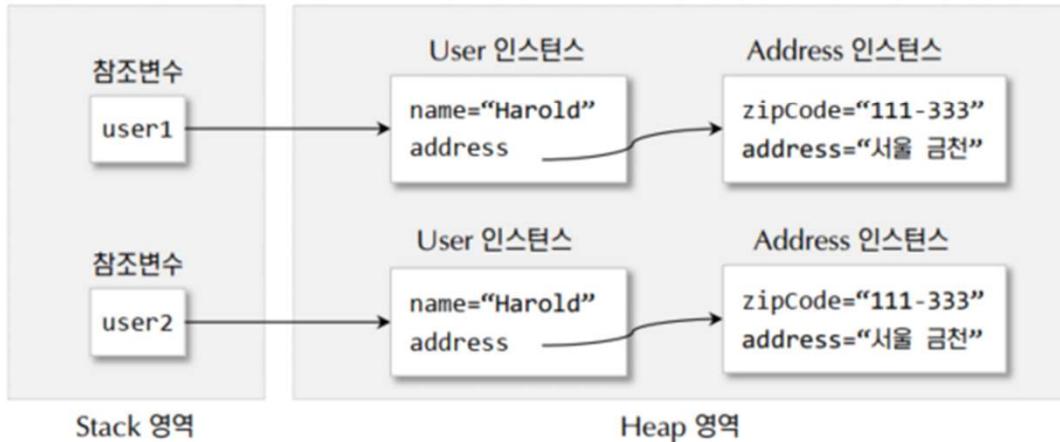
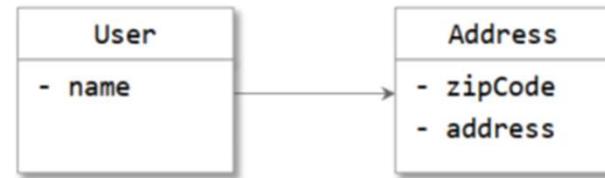


# Object Clone with interface (Cont.)

## ✓ Deep Copy

- User 클래스에서 clone() 메소드를 재정의하여, address 객체 복사합니다.
- User 클래스 내부의 address도 새로운 값이 복사되어서 참조하므로 같은 값을 참조하지 않게 됩니다.

```
@Override  
public Object clone() throws CloneNotSupportedException {  
    User clone = (User) super.clone();  
    if (address != null) {  
        clone.address = (Address) address.clone();  
    }  
    return clone;  
}
```



# Packages

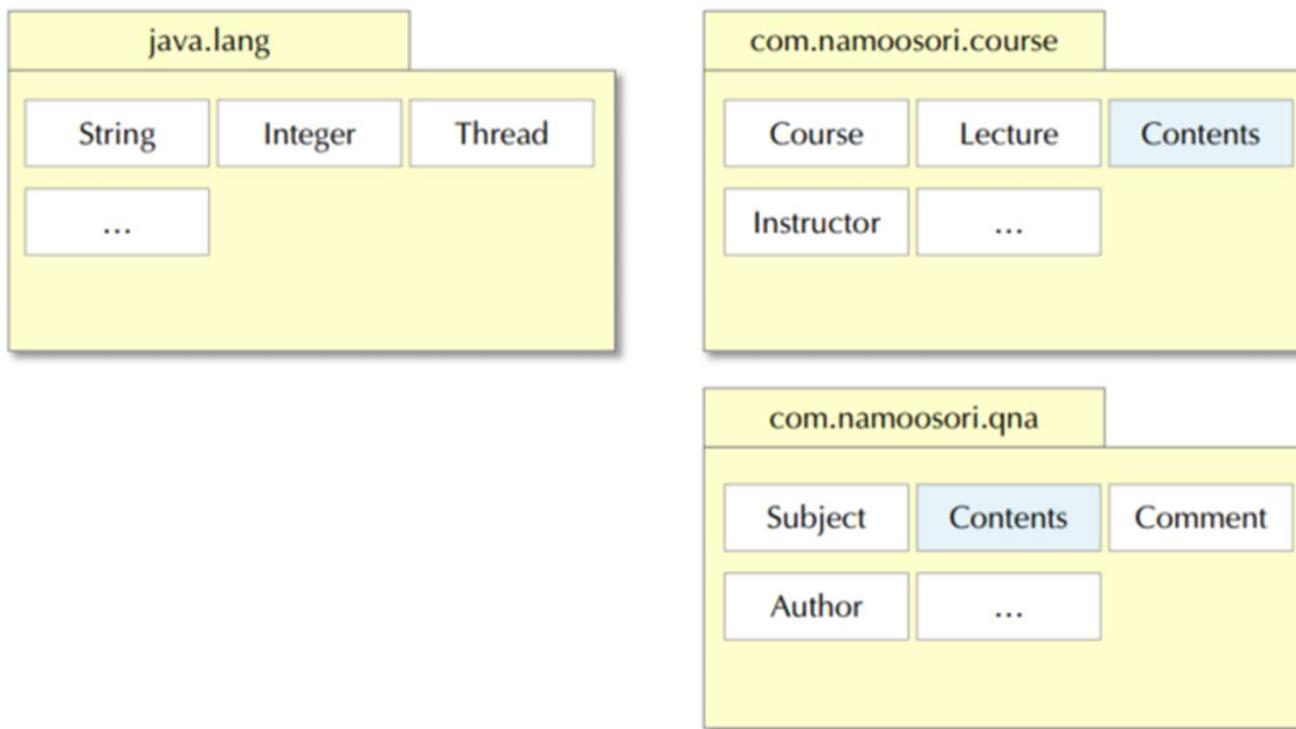
- You must specify package declaration at the beginning of the source file.
- You are permitted only one package declaration per source file.
- Package names must be hierarchical and separated by dots.

```
//class Employee of the Finance department for the ABC Company
package ABC.financeDept;
public class Employee {

}
```

# Packages (Cont.)

- ✓ Java 는 패키지를 이용하여 관련이 있는 클래스들을 묶어줍니다.
- ✓ 패키지를 사용하면 다른 곳에서 제공한 라이브러리와 작업 중인 것들을 분리하여 정리할 수 있습니다.
- ✓ 패키지를 사용하는 가장 중요한 이유는 클래스 이름에 대한 유일성을 보장할 수 있다는 점입니다.



# Directory Layout and Package

- Packages are stored in the directory tree containing the package name.

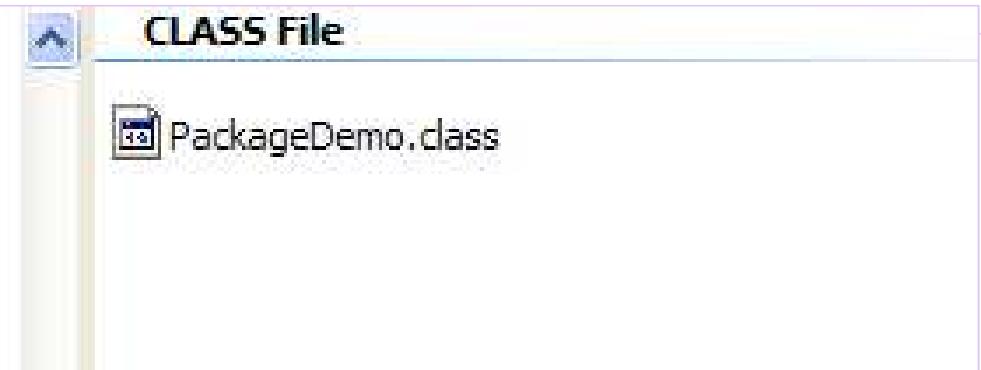
```
package ABC.financedept  
public class Employee {  
}
```

```
javac -d . Employee.java
```

## Packages (Cont.)

```
1 package kr.co.javaexpert.libs.j2se;
2
3 public class PackageDemo {
4     private String name;
5     public PackageDemo(String name){
6         this.name = name;
7     }
8     public void display(){
9         System.out.println("name = " + this.name);
10    }
11 }
```

```
C:\JavaRoom>javac -d C:\Temp PackageDemo.java
```



# The import Statement

- Tells the compiler where to find classes to use.
- Precedes all class declarations:

```
import ABC.financeDept.*;
public class Manager extends Employee {
    String department;
    Employee subordinates [];
}
```

# The import Statement (Cont.)

- ✓ 클래스는 자신의 패키지에 있는 모든 클래스와 다른 패키지의 모든 public 클래스들을 사용할 수 있습니다.
- ✓ 다른 패키지의 public 클래스를 사용하는 두 가지 방법입니다.
  - 패키지를 포함한 클래스명을 사용

```
java.util.Date today = new java.util.Date();
```

- import 문을 사용

```
import java.util.Date;  
...  
Date today = new Date();
```

- ✓ import 문

- import 문은 소스파일의 가장 상단에 위치하며, 포함시키는 클래스를 정의합니다.
- 와일드카드(\*)를 사용하여 특정 패키지의 모든 클래스를 포함시킬 수 있습니다. (예) java.util.\*; 단, 사용하려는 클래스가 두 곳 이상의 패키지에 포함된 경우 컴파일 에러가 발생합니다.

```
import java.util.*;  
import java.sql.*;  
  
// compile ERROR  
Date today = new Date();
```

```
import java.util.*;  
import java.sql.*;  
import java.util.Date;  
  
Date today = new Date();
```

# The **import** Statement (Cont.)

```
1 import kr.co.javaexpert.libs.j2se.PackageDemo;  
2  
3 public class ImportDemo {  
4     public static void main(String[] args) {  
5         PackageDemo pd = new PackageDemo("Duncan");  
6         pd.display();  
7     }  
8 }
```

C:\#JavaRoom>javac ImportDemo.java  
ImportDemo.java:1: error: package kr.co.javaexpert.libs.j2se does not exist  
import kr.co.javaexpert.libs.j2se.PackageDemo;  
  
ImportDemo.java:5: error: cannot find symbol  
 PackageDemo pd = new PackageDemo("Duncan");  
 ^  
 symbol: class PackageDemo  
 location: class ImportDemo  
ImportDemo.java:5: error: cannot find symbol  
 PackageDemo pd = new PackageDemo("Duncan");  
 ^  
 symbol: class PackageDemo  
 location: class ImportDemo  
3 errors

# CLASSPATH Setting

- Placing java class files to another directory.

- **javac.exe -d <directory>**

```
C:\JavaRoom>javac -d C:\temp Test.java
```

- Referencing another directory java class.

- Windows Platform

- **java.exe -classpath .;<directory>**

- Linux/Unix/Mac Platform

- **java.exe -classpath .:<directory>**

```
C:\JavaRoom>java -classpath .;C:\Temp ClassPathDemo  
name = Duncan
```

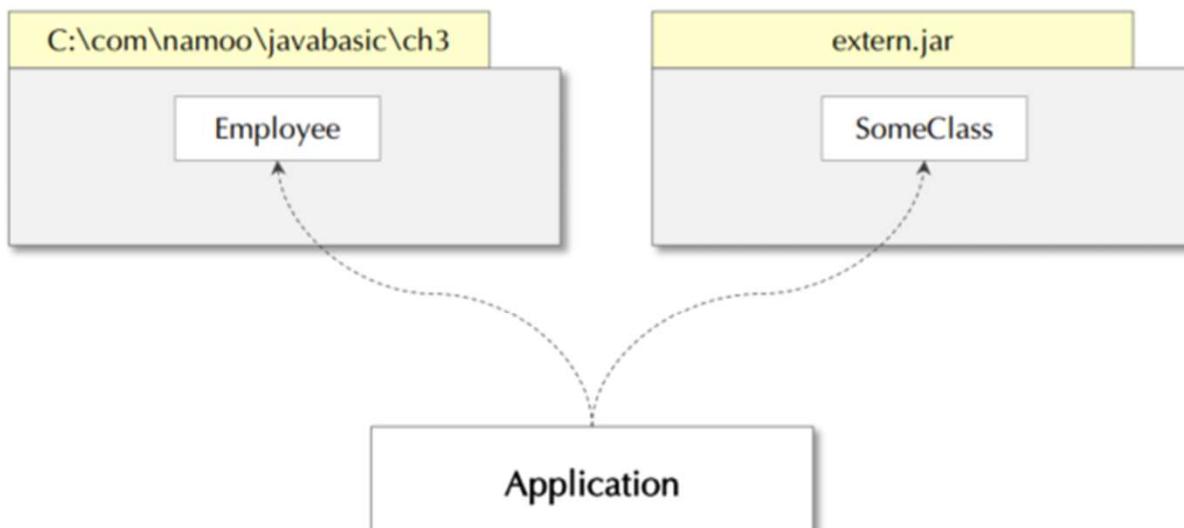
# CLASSPATH Setting (Cont.)

## ✓ 클래스패스(Class Path)

- 클래스는 파일시스템의 서브 디렉토리 안에 존재하거나, JAR 파일 내에 존재할 수 있습니다. 프로그램 사이에서 클래스가 공유되려면 먼저, 사용하는 클래스에 대한 클래스 패스를 명시해야 합니다.

## ✓ 클래스 패스와 현재 디렉토리(.)

- 클래스 패스가 명시되지 않으면 디폴트로 현재 디렉토리(.)가 클래스 패스에 포함됩니다.
- 그러나 하나라도 클래스패스가 명시되는 경우 현재 디렉토리는 디폴트로 포함되지 않으므로 반드시 추가해 주어야 합니다.



애플리케이션이 동작하려면 클래스패스가 설정되어 있어야 합니다.

# CLASSPATH Setting (Cont.)

## ✓ 클래스 패스를 설정하는 방법

- java 명령어의 –classpath (또는 –cp) 옵션 사용
- CLASSPATH 환경변수 사용

## ✓ -classpath 옵션 사용하기

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg
```

## ✓ CLASSPATH 환경변수 설정

- Bourne Again shell (bash)

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

- C shell

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

- Windows shell

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

## The **import** Statement (Cont.)

```
1 import kr.co.javaexpert.libs.j2se.PackageDemo;  
2  
3 public class ImportDemo {  
4     public static void main(String[] args) {  
5         PackageDemo pd = new PackageDemo("Duncan");  
6         pd.display();  
7     }  
8 }
```

```
C:\JavaRoom>javac -cp .;C:\Temp ImportDemo.java
```

```
C:\JavaRoom>java -cp .;C:\Temp ImportDemo  
name = Duncan
```

# The **import** Statement (Cont.)

```
C:\Temp>jar cvf mylib.jar kr/
```

```
added manifest  
adding: kr/(in = 0) (out= 0)(stored 0%)  
adding: kr/co/(in = 0) (out= 0)(stored 0%)  
adding: kr/co/com/(in = 0) (out= 0)(stored 0%)
```



Java 9 higher not work.

```
C:\JavaRoom>javac ImportDemo.java
```

```
C:\JavaRoom>java ImportDemo  
name = Duncan
```

File	Size
bridge.jar	83 KB
laf.jar	9 KB
jaccess.jar	43 KB
locatedata.jar	1,001 KB
sunec.jar	16 KB
sunjce_provider.jar	194 KB
sunmscapi.jar	30 KB
sunkcs11.jar	233 KB
zipfs.jar	68 KB
mylib.jar	2 KB
meta-index	1 KB

## Static-Import-on-Demand Declaration

- Allows all accessible **static** members declared in the type named by a canonical name to be imported as needed.
- Syntax :

```
import static TypeName.*;
```

## Static-Import-on-Demand Declaration (Cont.)

- ✓ `import static` 문을 사용하면 정적 메소드나 필드를 클래스명 없이 사용할 수 있습니다.
- ✓ 마찬가지로 와일드 카드를 사용하여 해당 클래스의 모든 정적 요소를 포함시킬 수 있습니다.
- ✓ 그러나, 클래스명 없이 정적 요소를 사용하는 것은 코드가 명시적이지 않아 권장하지 않습니다.

```
import static java.lang.System.out;
. . .
out.println("Goodbye, World!"); // 예) System.out
exit(0); // 예) System.exit
```

```
import java.lang.Math;
. . .
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
```

```
import static java.lang.Math.*;
. . .
sqrt(pow(x, 2) + pow(y, 2));
```

StaticImport.java

## Static-Import-on-Demand Declaration (Cont.)

```
1 import static java.lang.Math.*;
2 import static java.lang.System.out;
3
4 public class StaticImportDemo {
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         int su = (int)(random() * 10 + 1);
8         out.printf("su = %d\n", su);
9     }
10 }
```

# Advanced Access Control

- ✓ 접근 제한자(access modifier)는 외부로 공개할 범위를 지정하는 것으로 필드, 메소드 그리고 클래스에 적용합니다.
- ✓ 접근 제한자를 지정하지 않으면, 패키지 범위를 갖는 default 접근제한자가 적용됩니다.
- ✓ Java는 4가지 접근 제한자를 제공합니다.

접근제한자	클래스 내부	동일 패키지	상속받은 클래스	그 밖의 영역	UML 표기
private	O	X	X	X	-
default (package)	O	O	X	X	~
protected	O	O	O	X	#
public	O	O	O	O	+

# Advanced Access Control (Cont.)

- ✓ 접근 제한자에 따라 클래스 요소(클래스, 필드, 메소드)의 접근 범위는 달라집니다.
  - public으로 선언된 요소는 모든 클래스에서 접근할 수 있습니다.
  - private으로 선언된 요소는 동일한 클래스 내에서만 접근할 수 있습니다.
- ✓ 접근 제한자를 생략하면, 디폴트로 패키지 범위 접근 제한자가 적용됩니다.
  - 클래스의 경우 접근 제한자를 생략하여 패키지 내에서만 접근하도록 하는 것이 합리적일 수 있습니다.
  - 하지만, 데이터 필드의 경우 접근 제한자를 생략하는 것은 캡슐화를 깨는 결과를 가져옵니다.
  - 캡슐화가 깨진 java.awt 패키지의 Window 클래스  
캡슐화를 깨고 악의적인 문자열을 주입시킬 위험이 있습니다

```
package java.awt;
public class Window extends Container
{
    String warningString;
    . . .
}
```

참고. JDK 1.2부터 클래스로더는 사용자가 정의한 클래스가 java로 시작하는 패키지에 있으면 로딩을 어용하지 않습니다.

## Class(**static**) Variables

- Are shared among all instances of a class.

```
public class Count {  
    private int serialNumber;  
    public static int counter = 0;  
  
    public Count() {  
        counter++ ;  
        serialNumber = counter ;  
    }  
}
```

## Class(**static**) Methods

- You can invoke **static** method without any instance of the class to which it belongs.

```
1 class Test{  
2     public static void main(String[] args) {  
3         Math m = new Math();  
4         double d = m.random();  
5     }  
6 }
```

```
----- Java Compiler -----  
Test.java:3: Math() has private access in java.lang.Math  
        Math m = new Math();  
                           ^  
1 error
```

## **static Initializers**

- A class can contain code in a *static block* that does not exist within a method body.
- Static block code executes only once, when the class is loaded.
- A static block is usually used to initialize static (class) attributes.

## static Initializers (Cont.)

```
1 class Test{  
2     public static final int SU;  
3     public static double interest;  
4     static {  
5         SU = 5; //constant initialization  
6         interest = 0.35; //static variable initialization  
7     }  
8     public static void main(String[] args) {  
9     }  
10    }  
11 }
```

# The **final** Keyword

- You cannot subclass a **final** class.
  - e.g. **String**, **System**, **Math** class etc...
- You cannot override a **final** method.
  - e.g. **Math.random()**
- A **final** variable is a constant.
  - e.g. **Math.PI**

# Modifiers Review

	Available Modifiers
class	public,(default), final, abstract
method	public, protected, private, (default), final, abstract, static
member variable	public, protected, private, (default), final, static
local variable	final

- It cannot use **static** and **abstract** together in method.
- It cannot use **final** and **abstract** together in class.
- Access modifier of **abstract** method cannot be **private**.
- It cannot use **private** and **final** together in method.

# Deprecation

- Deprecation is the obsoleteness of class constructors and method calls.
- Obsolete methods and constructors are replaced by methods with a more standardized naming convention.
- When migrating code, compile the code with the  
– **deprecation** flag:

`javac –deprecation MyFile.java` or

`javac –Xlint:deprecation MyFile.java`

# Deprecation (Cont.)

```
1 class Test{  
2     public static void main(String[] args){  
3         java.util.Date now = new java.util.Date();  
4         int year = now.getYear();  
5     }  
6 }
```

----- Java Compiler -----

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

```
C:\JavaRoom>javac -deprecation Test.java  
Test.java:4: warning: [deprecation] getYear() in java.util.Date has been depreca  
ted
```

```
        int year = now.getYear();  
               ^
```

```
1 warning
```

```
C:\JavaRoom>javac -Xlint:deprecation Test.java  
Test.java:4: warning: [deprecation] getYear() in java.util.Date has been depreca  
ted
```

```
        int year = now.getYear();  
               ^
```

```
1 warning
```

## Inner Classes

- Added to JDK1.1
- Allow a class definition to be placed inside another class definition.
- Group classes that logically belong together.
- Have access to their enclosing class's scope.
- Nested Class
- Member Class
- Local Class
- Anonymous Class

# Nested Class

- Must be declared static class.
- Nested class's instance have no any relationship Outer class's instance directly.
- Nested class's method is referenced only self's member and Outer class's static member.

```
class NestedTelevision {  
    private static int person = 5;          //static member  
    static class Television {    //Nested class  
        int inch = 20 ;           //Nested class's member  
        public void showTelevision() {  
            person ;  
            inch ;  
        }  
    }  
}
```

```
NestedTelevision.Television tv = new NestedTelevision.Television();  
tv.showTelevision();
```

# Member Class

- Is a member of Outer class.
- Is referenced self's member and Outer class's all member.

```
class MemberTelevision {  
    private int person = 5;          //member variable  
    public class Television {      //Member class  
        int inch = 20 ;           //Member class's member  
        public void showTelevision() {  
            person ;  
            inch ;  
        }  
    }  
}
```

```
MemberTelevision member = new MemberTelevision();  
MemberTelevision.Television tv = member.new Television();  
tv.showTelevision();
```

# Local Class

- Be placed within a Method.
- Is declared only within a method and is used within method. Therefore this class is temporary class. Nothing is declared or used at the outer class.
- Accessed only final local variable.
- Cannot have any access modifier.

```
class LocalTelevision {  
    public void showTV(final int person) {  
        class Television { //local class  
            int inch = 20 ;  
            public void showTelevision() {  
                person ;  
                inch ;  
            }  
            Television tv = new Television();  
            tv.showTelevision();  
        }  
    }  
}
```

## enum type

- Are implicitly **final**, because they declare constants that should not be modified.
- **enum** constants are implicitly **static**.
- Any attempt to create an object of an **enum** type with operator **new** results in a compilation error.
- The **enum** constants can be used anywhere constants can be used
  - In the **case** labels of **switch** statements
  - To control enhanced **for** statements.

## enum type (Cont.)

### ✓ 열거형이란 유효한 값(상수)들의 집합을 의미합니다.

- Java 1.5가 나오기 전까지는 열거형을 표현하기 위하여 static final 을 이용한 상수를 선언하거나, 인터페이스 기반 상수를 선언하는 방식으로 열거형을 구현했습니다.
- Java 1.5 부터는 enum 키워드를 사용하여, 보다 쉽고 안전하게 열거 값을 정의할 수 있습니다.

```
public enum Size {  
    SMALL,  
    MEDIUM,  
    LARGE,  
    EXTRA_LARGE  
};
```

*Size.java*

### ✓ enum과 java.lang.Enum

- enum 키워드를 사용하여 새로운 열거형을 선언하는 것은 간접적으로 java.lang.Enum을 확장하는 것입니다.

## **enum type (Cont.)**

### ■ Syntax :

```
Class_Modifier enum identifier{  
    enumConstants1, enumConstants2,...,enumConstantsn  
}
```

## enum type (Cont.)

```
2 public class Example {  
3     public enum Season{  
4         WINTER, SPRING, SUMMER, FALL  
5     }  
6     public static void main(String[] args) {  
7         // TODO Auto-generated method stub  
8         Season s = Season.SUMMER;  
9         System.out.println("s = " + s);  
10        System.out.println("SPRING : " + Season.SPRING);  
11    }  
12 }
```

```
C:\JavaRoom\JavaTest>dir Ex*.*  
Volume in drive C has no label.  
Volume Serial Number is 2486-E652
```

Directory of C:\JavaRoom\JavaTest

S = SUMMER  
SPRING : SPRING

## enum type (Cont.)

```
2 public enum Lesson {  
3     JAVA, XML, EJB  
4 }
```

```
2 public class Example1 {  
3     public static void main(String[] args) {  
4         // TODO Auto-generated method stub  
5         Lesson le = Lesson.EJB;  
6         System.out.println("le = " + le);  
7         System.out.println("XML : " + Lesson.XML);  
8     }  
9 }
```

```
le = EJB  
XML : XML
```

09/19/2006 10:36 PM

1,063 Lesson.class

Directory of C:\JavaRoom\JavaTest

09/19/2006 10:39 PM

803 Example1.class

## enum type (Cont.)

```
2 public enum State {  
3     INIT, OPENED, CLOSED;  
4 }  
  
2 public class EnumDemo {  
3     private State state;  
4  
5     private void setState(State state){  
6         this.state = state;  
7     }  
8  
9     public void open(){  
10        this.setState(State.OPENED);  
11    }  
12  
13    public void opening(){  
14        this.setState(State.OPENING); //Compile Error  
15    }  
16 }
```

# enum type (Cont.)

All Classes  
[Lesson](#)

Package [Class](#) Tree Deprecated Index Help  
PREV CLASS NEXT CLASS  
SUMMARY: NESTED | [ENUM CONSTANTS](#) | FIELD | [METHOD](#)

FRAMES NO FRAMES  
DETAIL: [ENUM CONSTANTS](#) | FIELD | [METHOD](#)

## Enum Lesson

java.lang.Object  
└ java.lang.Enum<[Lesson](#)>  
└ [Lesson](#)

All Implemented Interfaces:  
[java.io.Serializable](#), [java.lang.Comparable<Lesson>](#)

---

```
public enum Lesson
extends java.lang.Enum<Lesson>
```

## Enum Constant Summary

[EJB](#)  
[JAVA](#)  
[XML](#)

## Method Summary

static <a href="#">Lesson</a>	<a href="#">valueOf</a> (java.lang.String name)	Returns the enum constant of this type with the specified name.
static <a href="#">Lesson</a> []	<a href="#">values()</a>	Returns an array containing the constants of this enum type, in the order they're declared.

## Methods inherited from class `java.lang.Enum`

[clone](#), [compareTo](#), [equals](#), [getDeclaringClass](#), [hashCode](#), [name](#), [ordinal](#), [toString](#), [valueOf](#)

## enum type (Cont.)

```
2 public class Example2 {  
3     public enum Season {  
4         WINTER, SPRING, SUMMER, FALL  
5     }  
6     public static void main(String[] args) {  
7         // TODO Auto-generated method stub  
8         for(Season s : Season.values())  
9             System.out.println(s);  
10    }  
11 }
```

WINTER  
SPRING  
SUMMER  
FALL

## enum type (Cont.)

- ✓ 열거형 값을 비교할 때, if 문이나 switch 문을 사용합니다.
- ✓ == 또는 .equals() 메소드를 사용하여 enum 인스턴스를 비교합니다.
- ✓ 열거형 값은 “enum이름.상수명”의 형태로 접근합니다.

```
Size select = Size.SMALL; // 변수에 enum 값을 대입한다.  
if (select == Size.SMALL) System.out.println("It's Small!");  
if (select.equals(Size.LARGE)) System.out.println("It's Large!");  
  
Size size = Size.LARGE;  
switch (size) {  
    case SMALL: System.out.println("Small");  
    case MEDIUM: System.out.println("Medium");  
    case LARGE: System.out.println("Large");  
    case EXTRA_LARGE: System.out.println("Extra Large");  
}
```



이 코드의 수행결과는 무엇일까요?

EnumUse.java

## enum type (Cont.)

```
2 public class Example3 {  
3     public enum Season {  
4         WINTER, SPRING, SUMMER, FALL  
5     }  
6     public static void main(String[] args) {  
7         // TODO Auto-generated method stub  
8         Season s = Season.SUMMER;  
9         if(s instanceof Object){  
10             System.out.println(s.toString());  
11             System.out.println("OK! instanceof Object");  
12             System.out.println("Real Value is " + s.ordinal());  
13         }  
14         Season [] array = Season.values();  
15         System.out.println("array.length = " + array.length);  
16         for(Season s1 : array)  
17             System.out.println(s1 + " ==> " + s1.ordinal());  
18     }  
19 }
```

SUMMER

OK! instanceof Object

Real Value is 2

array.length = 4

WINTER ==> 0

SPRING ==> 1

SUMMER ==> 2

FALL ==> 3

## enum type (Cont.)

- ✓ enum 에는 생성자, 메소드, 변수 그리고 상수 별 클래스 본체를 추가할 수 있습니다.
- ✓ enum 은 일반 클래스와 유사하게 사용 가능합니다.

```
public class AdvancedEnum {  
  
    enum Names {  
        JERRY("Lead guitar") {  
            public String sings() { return "plaintively"; }  
        },  
        BOBBY("rhythm guitar") {  
            public String sings() { return "hoarsely"; }  
        },  
        PHIL("bass");  
  
        private String instrument;  
  
        Names(String instrument) {  
            this.instrument = instrument;  
        }  
        public String getInstrument() {  
            return this.instrument;  
        }  
        public String sings() {  
            return "occasionally";  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    for (Names n : Names.values()) {  
        System.out.print(n);  
        System.out.print(", instrument: " + n.getInstrument());  
        System.out.println(", sings : " + n.sings());  
    }  
}  
AdvancedEnum.java
```

## enum type (Cont.)

```
2 public class Example4 {  
3     private enum CoinColor {  
4         COPPER, NICKEL, SILVER  
5     }  
6     private static CoinColor color(Coin c){  
7         switch(c){  
8             case PENNY: return CoinColor.COPPER;  
9             case NICKEL: return CoinColor.NICKEL;  
10            case DIME:  
11            case QUARTER: return CoinColor.SILVER;  
12            default:  
13                throw new AssertionError("Unknown coin : " + c);  
14        }  
15    }  
16    public static void main(String[] args) {  
17        // TODO Auto-generated method stub  
18        for(Coin c : Coin.values())  
19            System.out.println(c + " : " + c.getValue() + "cent " + color(c));  
20    }  
21 }
```

```
1 public enum Coin{  
2     PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
3     Coin(int value) { //constructor  
4         this.value = value;  
5     }  
6     private final int value;  
7     public int getValue() { return this.value; }  
8 }
```

PENNY :	1cent	COPPER
NICKEL :	5cent	NICKEL
DIME :	10cent	SILVER
QUARTER :	25cent	SILVER