Object Orientation
Fourth Story

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/JavaSE

`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

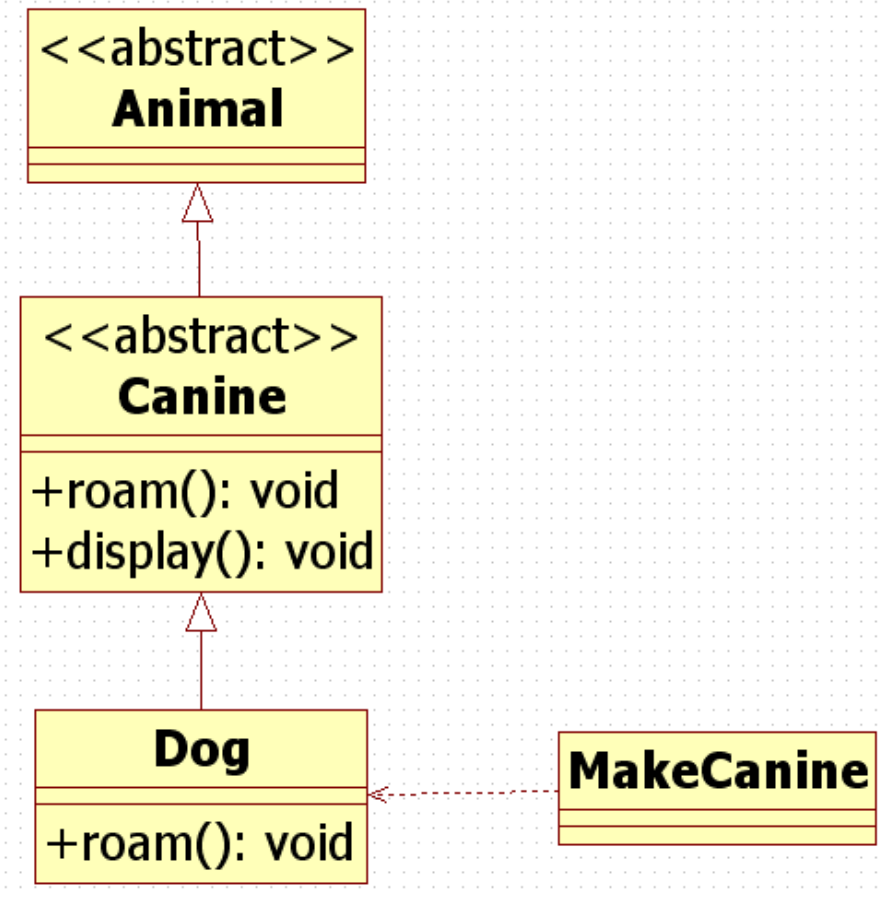
`abstract` Classes (Cont.)

```
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 }
```



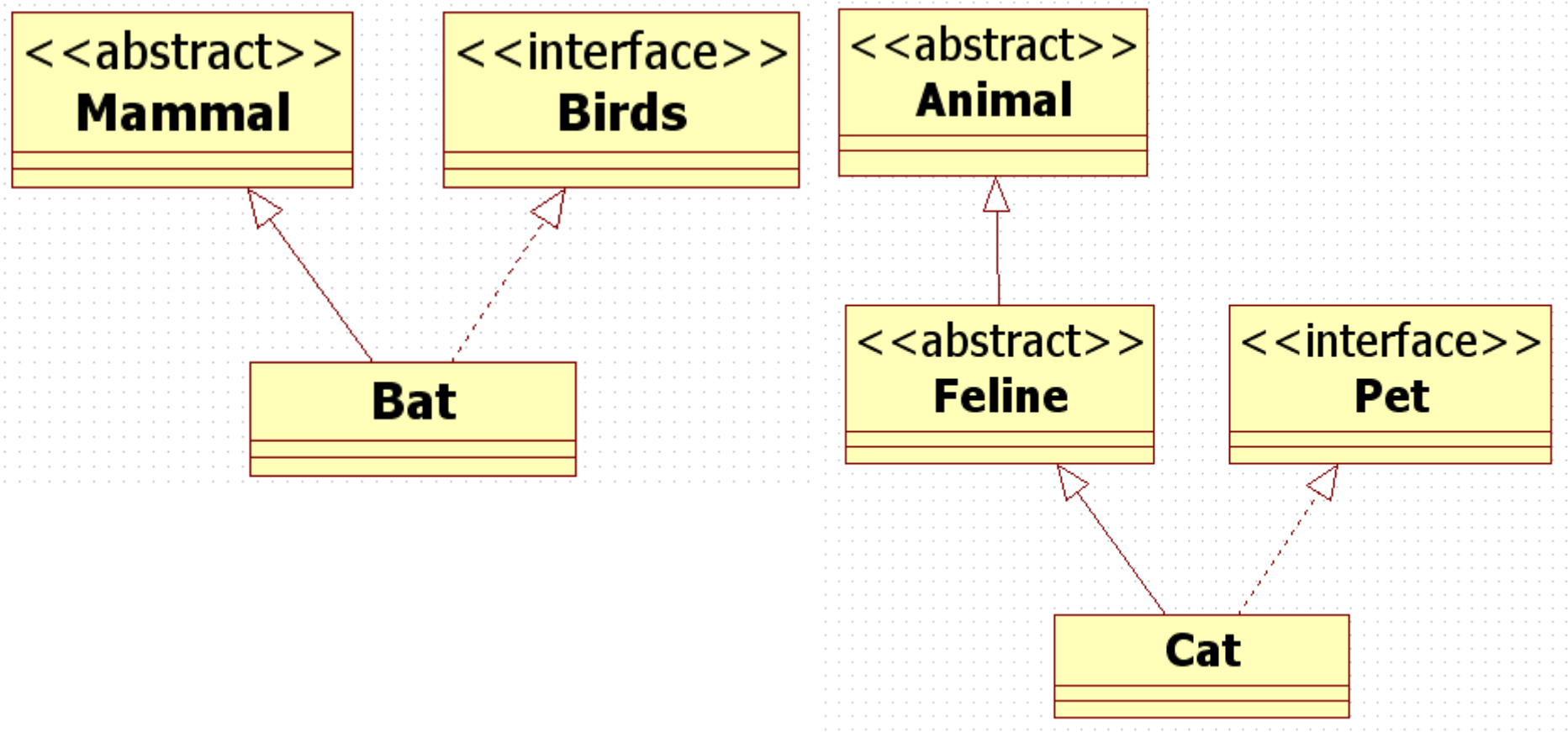
`interfaces`

- 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.)



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 Candyis 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...
```

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
```

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 {

 }
```

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.)

```
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
```

The `import` Statement (Cont.)

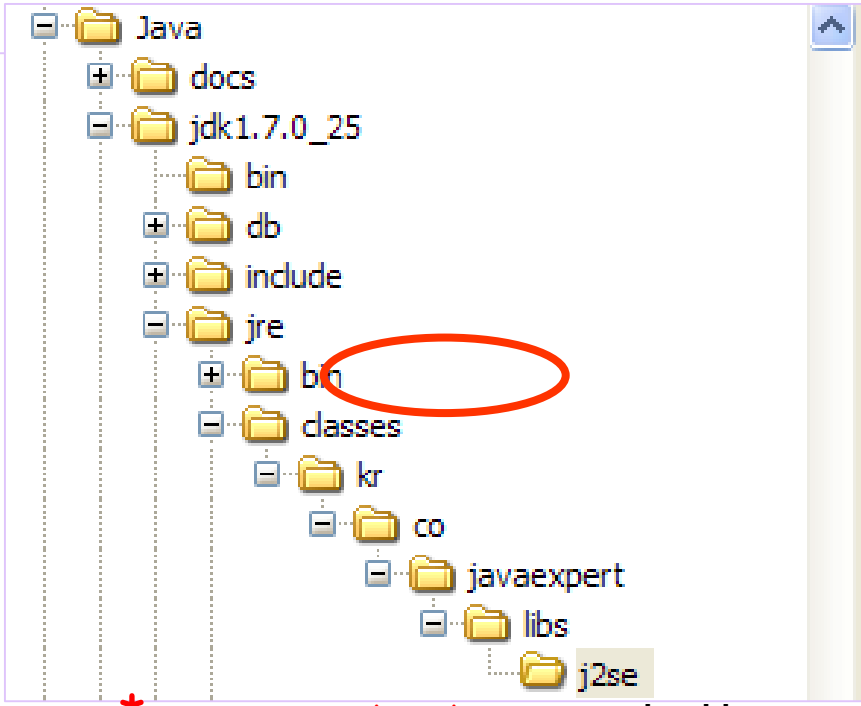
```
1  import kr.co.javaexpert.libs.j2se.PackageDemo;
2
3  public class ImportDemo {
4      public static void main(String[] args) {
5          PackageDemo  pd = new PackageDemo("Duncan");
6          pd.display();
7      }
8  }
```

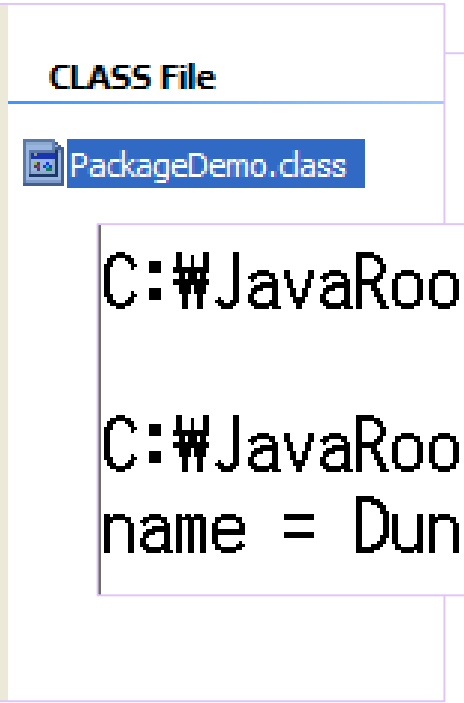
```
C:₩JavaRoom>javac -cp .;C:₩Temp ImportDemo.java

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

The `import` Statement (Cont.)

```
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

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

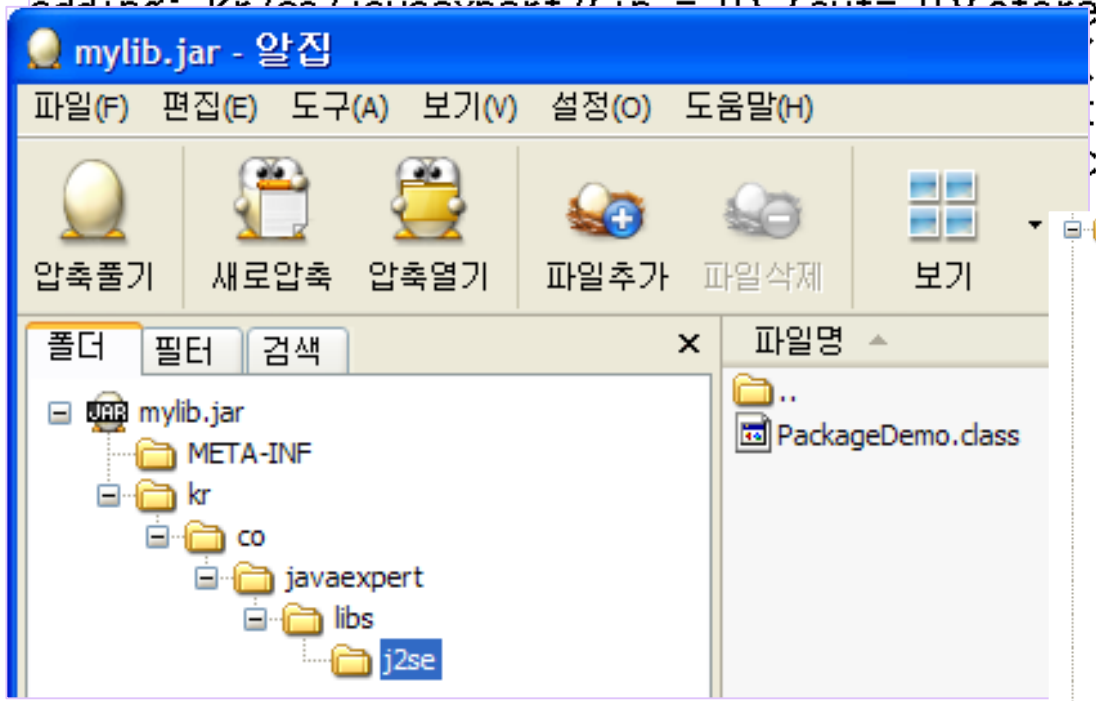
* `%JAVA_HOME%\jre\classes` should create manually because when JDK install is not created automatically.

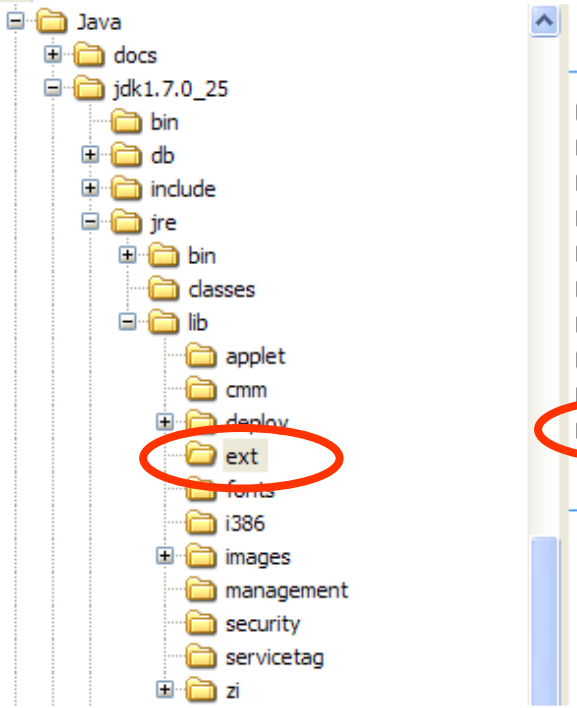
The `import` Statement (Cont.)

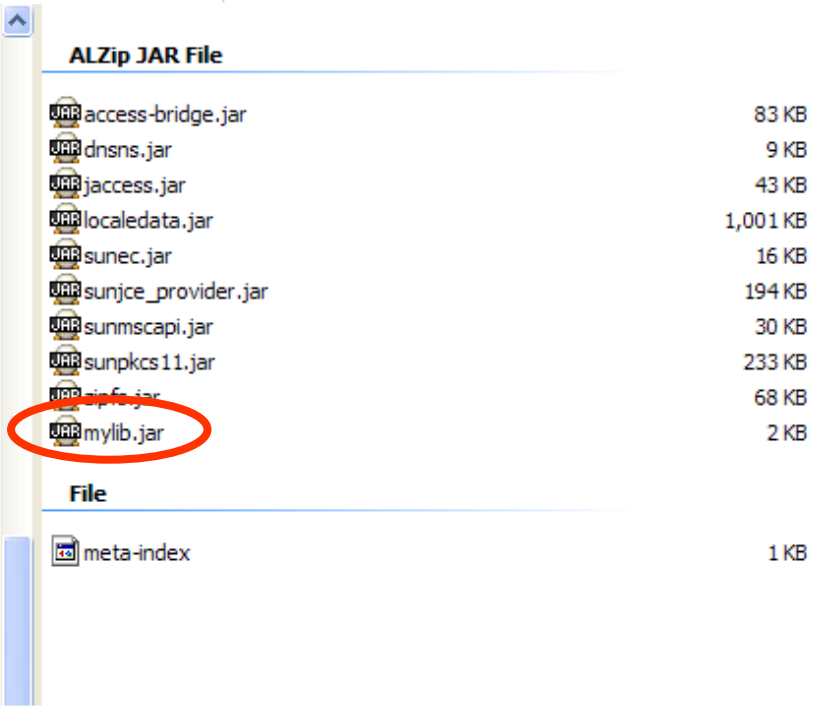
```
C:₩Temp>jar cvf mylib.jar kr/
added manifest
adding: kr/(in = 0) (out= 0)(stored 0%)
adding: kr/co/(in = 0) (out= 0)(stored 0%)
                                          d 0%)
                                          stored 0%)
                                          = 0)(stored
                                          lass(in = 652) (out= 402)(deflat
```

```
C:₩JavaRoom>javac ImportDemo.java

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







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.)

```
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

Modifier	Same class	Same Package	Subclass	Universe
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	
`default`	Yes	Yes		
`private`	Yes			

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 deprecated
                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 deprecated
                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.Televison 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.)

- 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 }
```

```
s = SUMMER
SPRING : SPRING
```

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

 Directory of C:\JavaRoom\JavaTest

09/19/2006  10:29 PM             1,220 Example$Season.class
09/19/2006  10:29 PM               863 Example.class
09/19/2006  10:29 PM               293 Example.java
               3 File(s)          2,376 bytes
               0 Dir(s)   8,548,823,040 bytes free
```

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 | FRAMES NO FRAMES
SUMMARY: NESTED | ENUM CONSTANTS | FIELD | METHOD | 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 Lesson	**valueOf**(java.lang.String name) Returns the enum constant of this type with the specified name.
static Lesson[]	**values**() 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.)

```
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! instnceof 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! instnceof Object
Real Value is 2
array.length = 4
WINTER ==> 0
SPRING ==> 1
SUMMER ==> 2
FALL ==> 3
```

enum type (Cont.)

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

```java
public class Example4 {
   private enum CoinColor {
      COPPER, NICKEL, SILVER
   }
   private static CoinColor color(Coin c){
      switch(c){
         case PENNY:  return CoinColor.COPPER;
         case NICKEL:   return CoinColor.NICKEL;
         case DIME:
         case QUARTER:  return CoinColor.SILVER;
         default:
            throw new AssertionError("Unknown coin : " + c);
      }
   }
   public static void main(String[] args) {
      // TODO Auto-generated method stub
      for(Coin c : Coin.values())
         System.out.println(c + " :      " + c.getValue() + "cent  " + color(c));
   }
}
```

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