Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture 8: Packages and Interfaces

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Packages

- Packages are containers for classes.
- Unique name had to be used for each class to avoid name collisions.
- Need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
    - (Imagine a small group of programmers fighting over who gets to use the name “Cricket” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Apple”)
- Java provides a mechanism for partitioning the class name space into more manageable chunks.
- This mechanism is the package.
- The package is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are exposed only to other members of the same package.
- This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

- To create a package is quite easy simply include a **package** command as the first statement in a Java source file.
- This is the general form of the **package** statement:

  package *pkg;*

  Here, *pkg* is the name of the package.
- For example, the following statement creates a package called **MyPackage**:

  package MyPackage;
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.
- While the default package is fine for short, sample programs, it is inadequate for real applications.
- Most of the time, you will define a package for your code.

Defining a Package

- Java uses file system directories to store packages.
  - For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.
- The **package** statement simply specifies to which package the classes defined in a file belong.
- It does not exclude other classes in other files from being part of that same package.
- Most real-world packages are spread across many files.
- You can create a hierarchy of packages.
- To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:
  package *pkg1*[.*pkg2*[.*pkg3*]];
- A package hierarchy must be reflected in the file system of your Java development system.
  - For example, a package declared as packagejava.awt.image; needs to be stored in **java\awt\image** folder

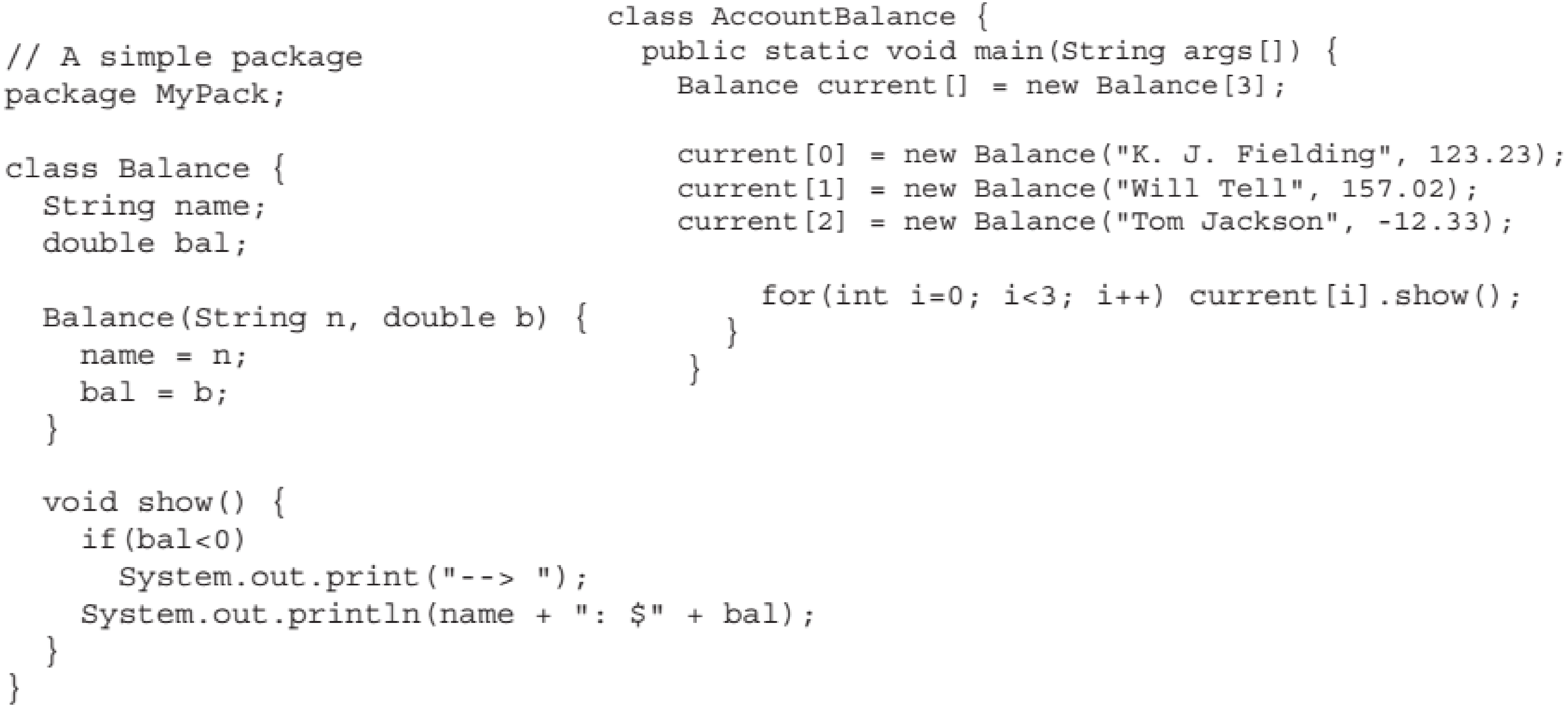
Finding Packages and CLASSPATH

- For example, consider the following package specification:

  package MyPack

- In order for a program to find **MyPack**, one of three things must be true.
  - Either the program can be executed from a directory immediately above **MyPack**, or
  - The **CLASSPATH** must be set to include the path to **MyPack**, or
  - The **-classpath** option must specify the path to **MyPack** when the program is run via **java**.
- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to* **MyPack**.
- If the path to **MyPack** is \home\MyPrograms\Java\MyPack
  then the class path to **MyPack** is \home\MyPrograms\Java
- The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory.

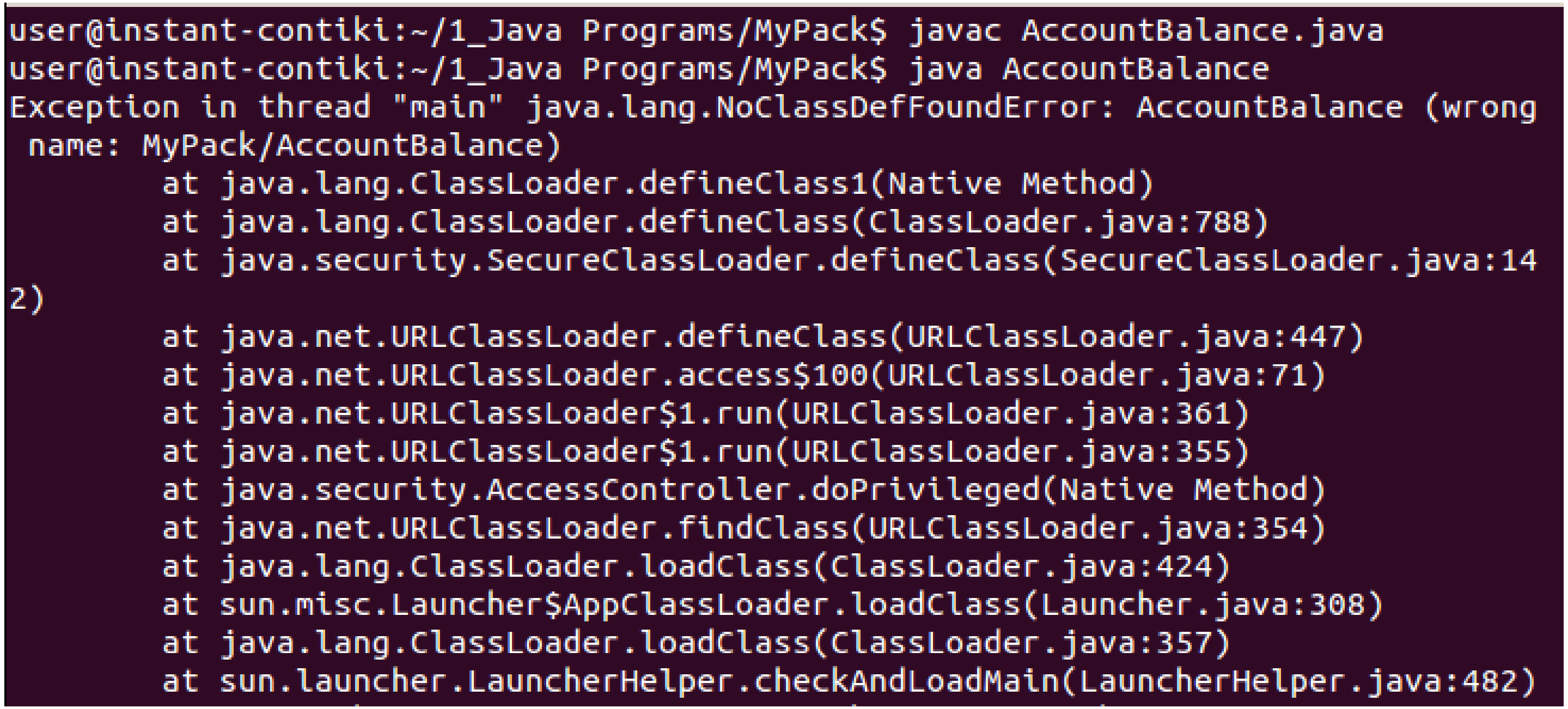
Example #1



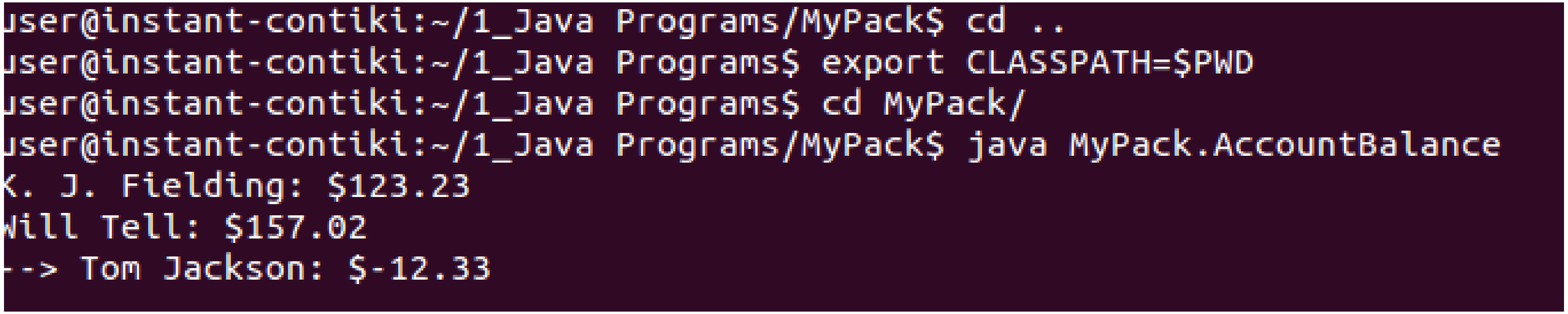
```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

```
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

      for(int i=0; i<3; i++) current[i].show();
    }
  }
```

Output



```
user@instant-contiki:~/1_Java Programs/MyPack$ javac AccountBalance.java
user@instant-contiki:~/1_Java Programs/MyPack$ java AccountBalance
Exception in thread "main" java.lang.NoClassDefFoundError: AccountBalance (wrong
 name: MyPack/AccountBalance)
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:788)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:14
2)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:447)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:482)
```



```
user@instant-contiki:~/1_Java Programs/MyPack$ cd ..
user@instant-contiki:~/1_Java Programs$ export CLASSPATH=$PWD
user@instant-contiki:~/1_Java Programs$ cd MyPack/
user@instant-contiki:~/1_Java Programs/MyPack$ java MyPack.AccountBalance
K. J. Fielding: $123.23
Will Tell: $157.02
--> Tom Jackson: $-12.33
```

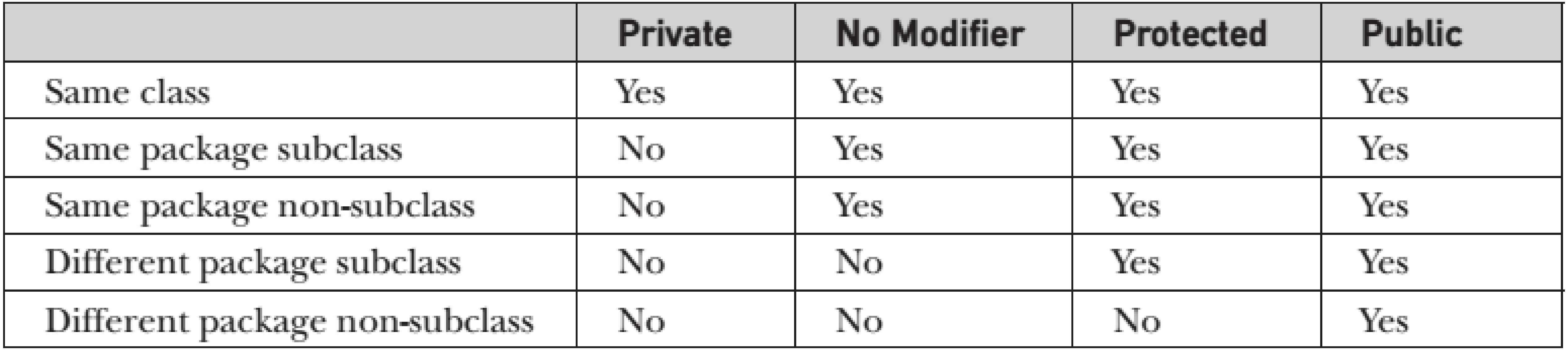
Access Protection

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- The class is Java’s smallest unit of abstraction.
- Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to
  produce the many levels of access required by these categories.

Access Protection

- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
  - When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
  - This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

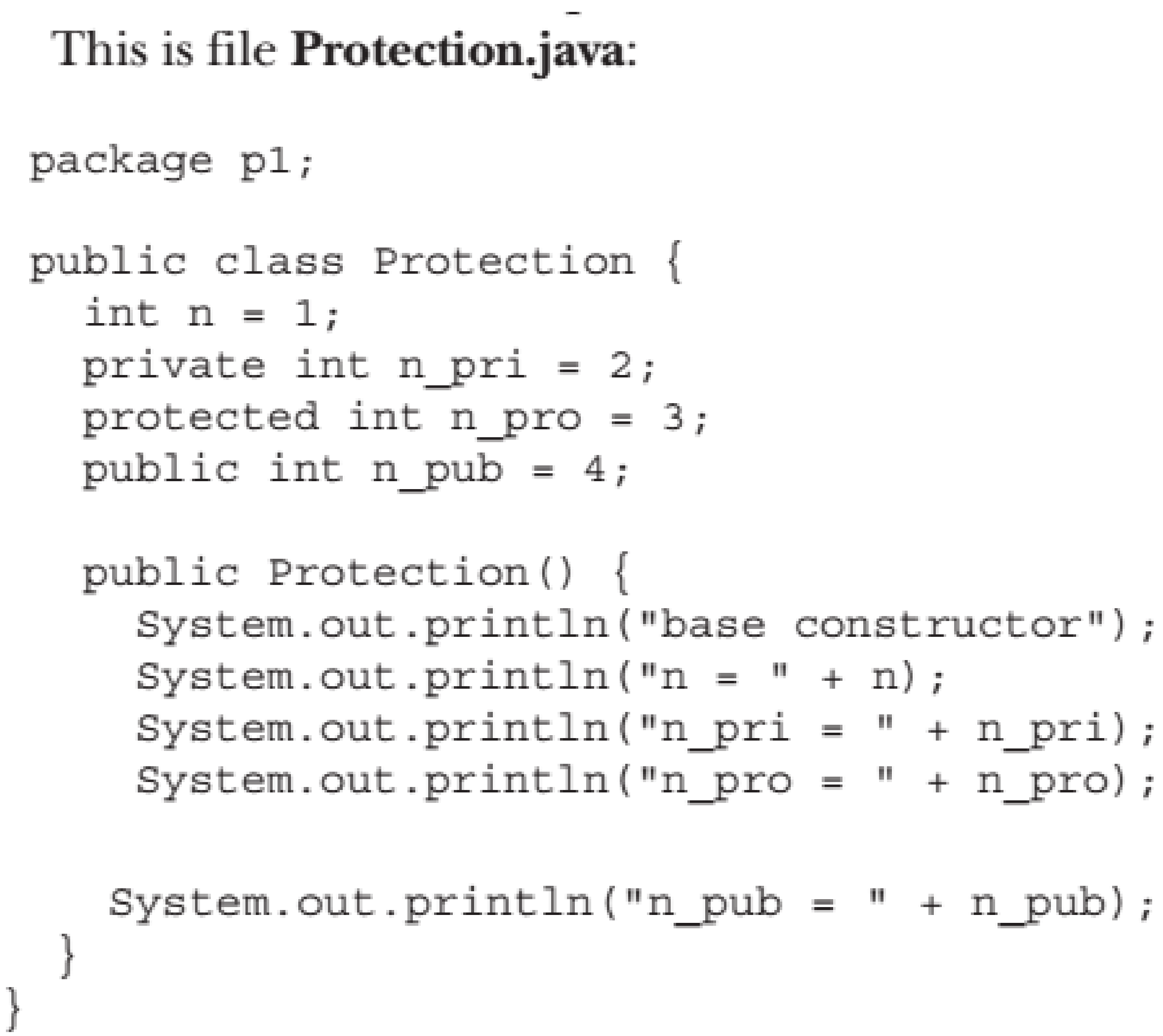
Class Member Access



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

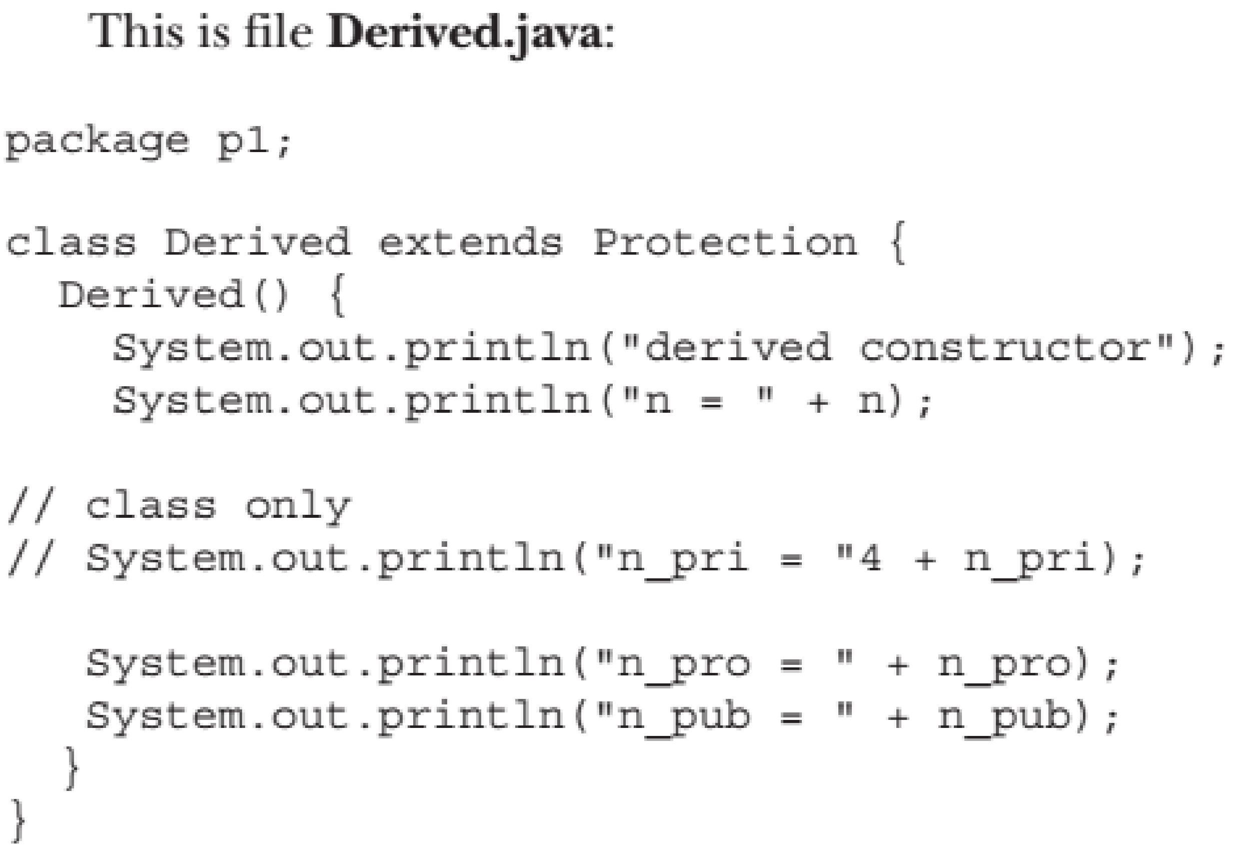
This table applies only to members of classes.
A non-nested class has only two possible access levels: default and public.

Example #2 to demonstrate An Access Control



This is file **Protection.java**:

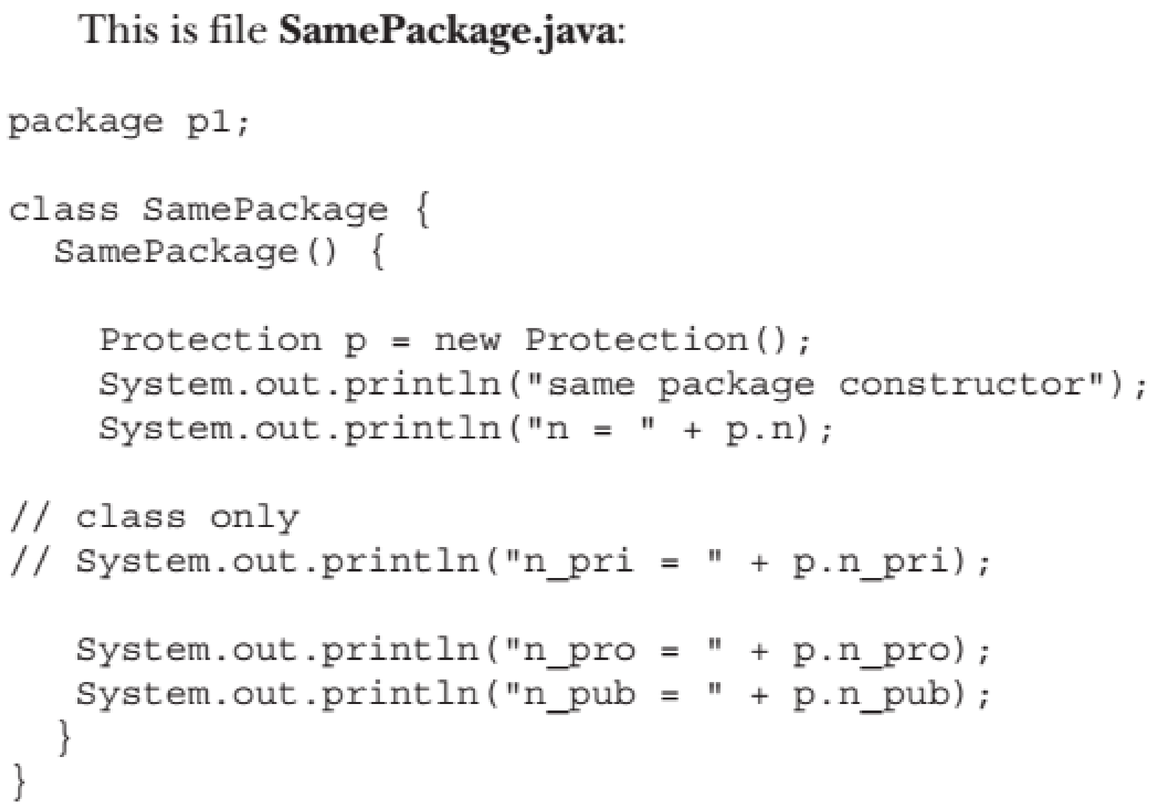
```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);


   System.out.println("n_pub = " + n_pub);
 }
}
```



This is file **Derived.java**:

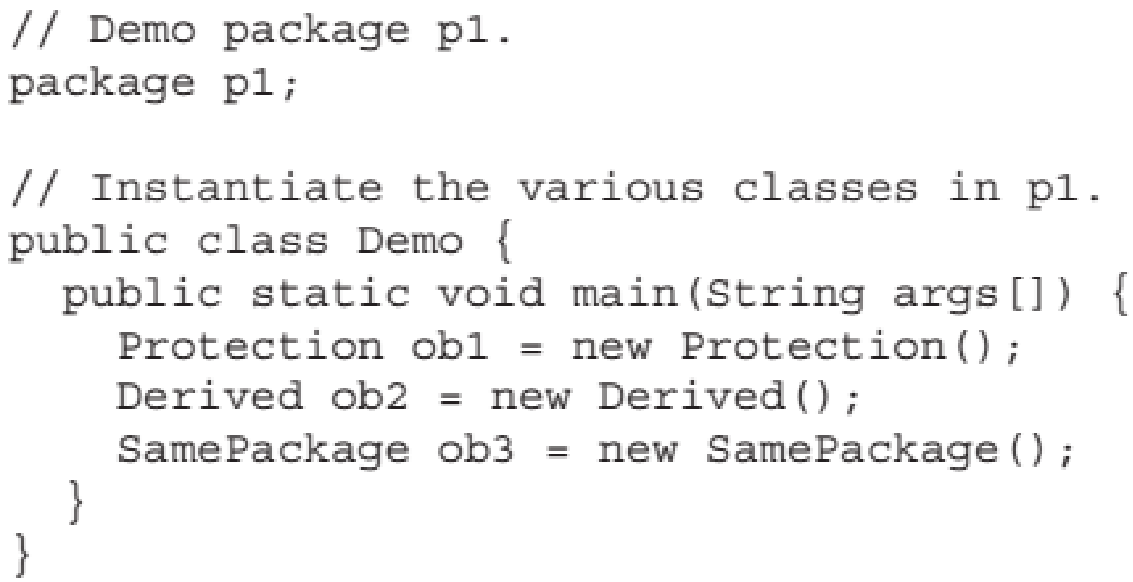
```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

// class only
// System.out.println("n_pri = "4 + n_pri);

   System.out.println("n_pro = " + n_pro);
   System.out.println("n_pub = " + n_pub);
  }
}
```

Example #2 to demonstrate An Access Control



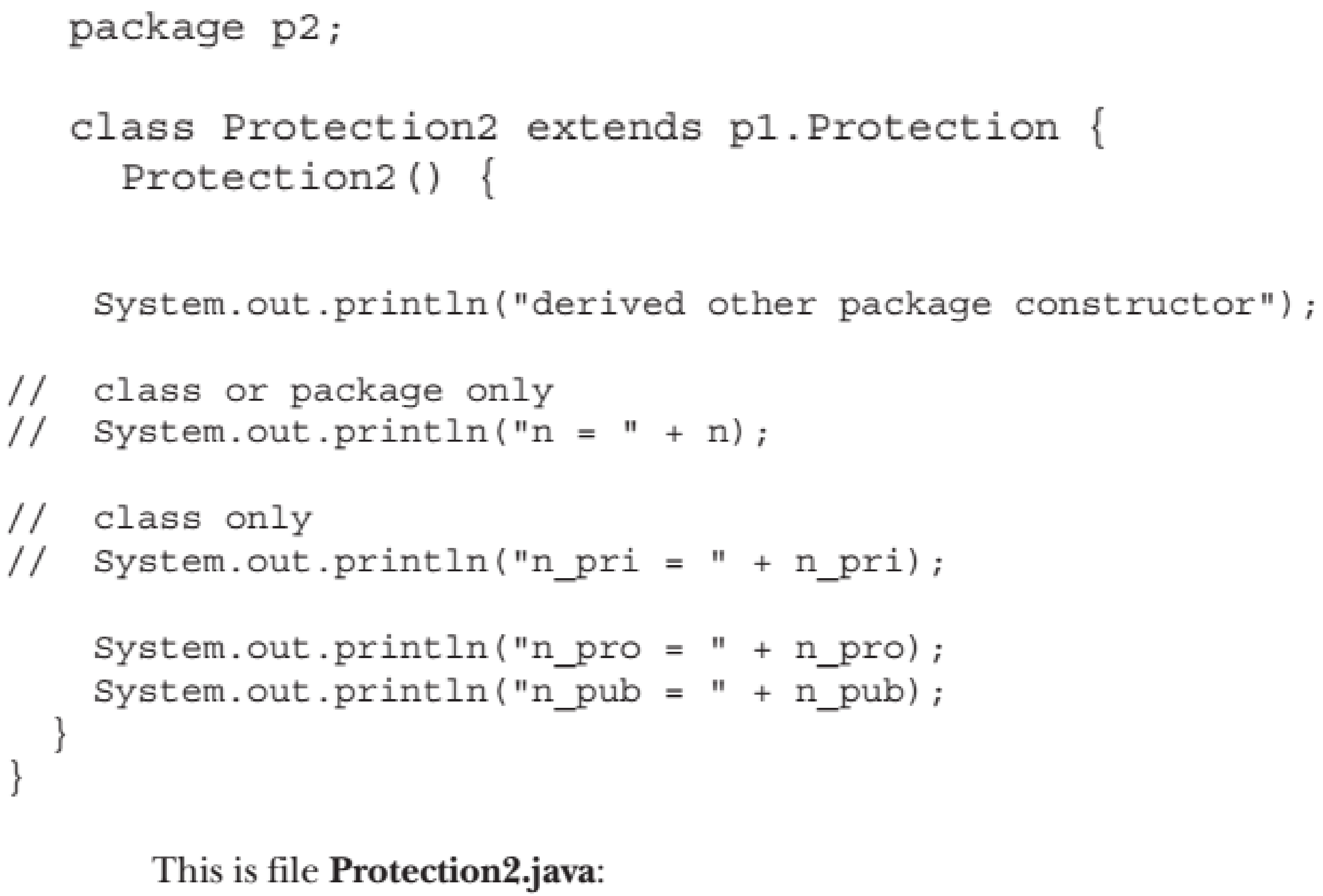
This is file **SamePackage.java**:

```
package p1;

class SamePackage {
  SamePackage() {

    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

   System.out.println("n_pro = " + p.n_pro);
   System.out.println("n_pub = " + p.n_pub);
  }
}
```



```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

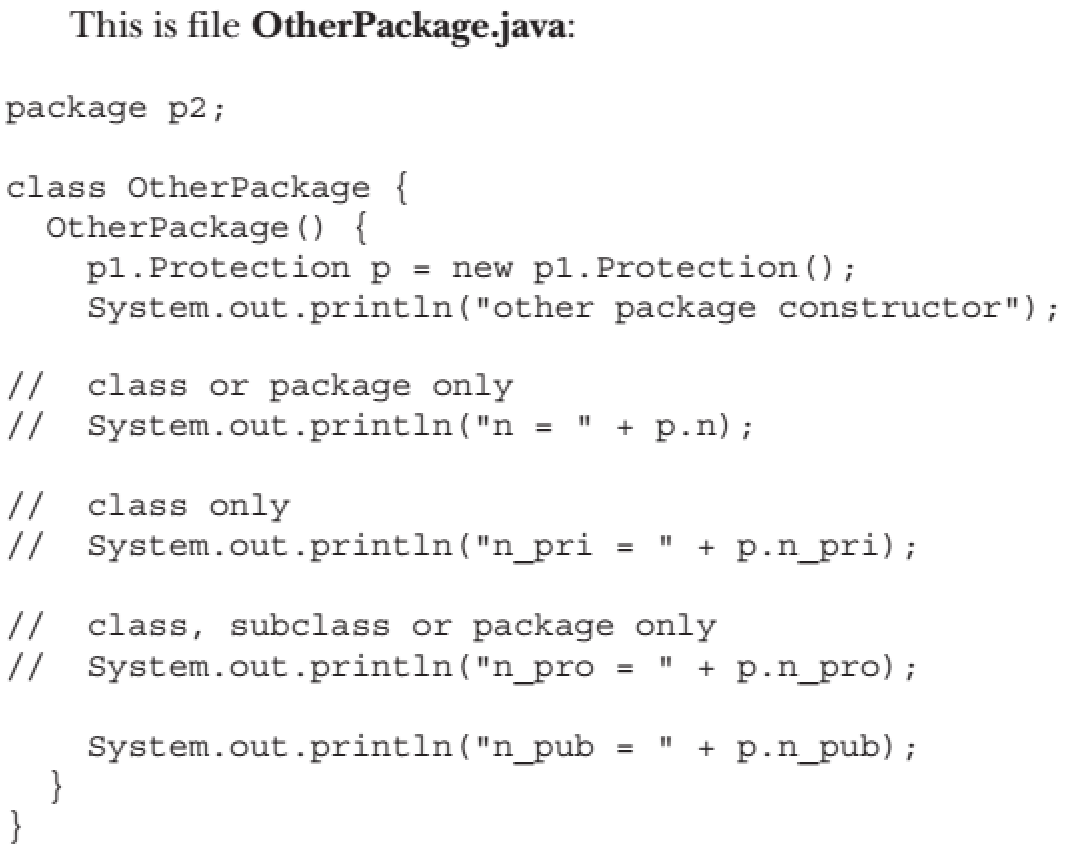
Example #2 to demonstrate An Access Control



```
  package p2;

  class Protection2 extends p1.Protection {
    Protection2() {


     System.out.println("derived other package constructor");

//   class or package only
//   System.out.println("n = " + n);

//   class only
//   System.out.println("n_pri = " + n_pri);

     System.out.println("n_pro = " + n_pro);
     System.out.println("n_pub = " + n_pub);
   }
}
```

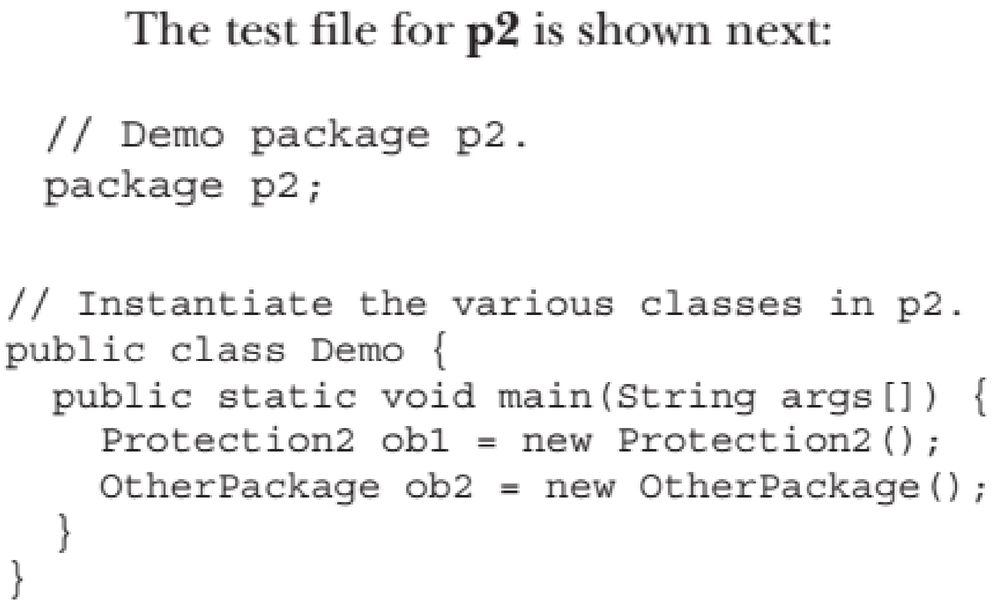
This is file **Protection2.java**:

Example #2 to demonstrate An Access Control



This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```



The test file for **p2** is shown next:

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Importing Packages

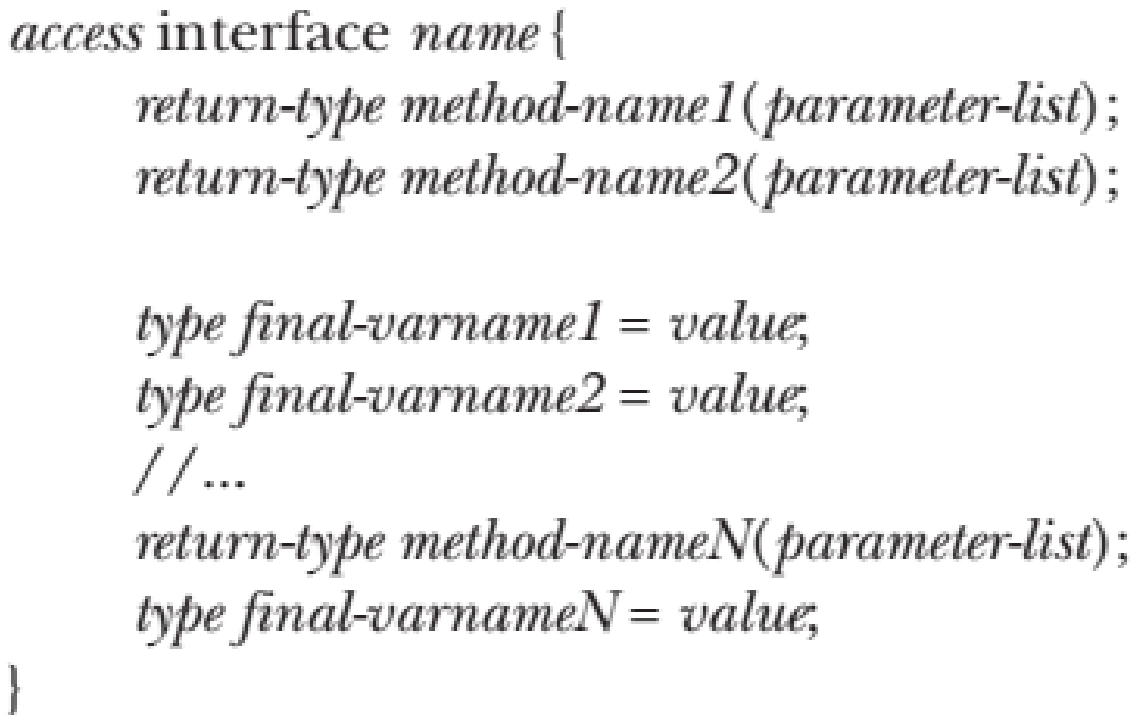
- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- This is the general form of the **import** statement:
  import *pkg1* [.*pkg2*].(*classname* | *);
- Examples
  - import java.util.Date;
  - import java.io.*;
- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs.
  - import java.lang.*;

Interfaces

- Using the keyword **interface**, you can fully abstract a class’ interface from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- In practice, this means that you can define interfaces that don’t make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an **interface**.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must provide the complete set of methods required by the interface.
- However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

Defining an Interface

- An interface is defined much like a class. This is a simplified general form of an interface:



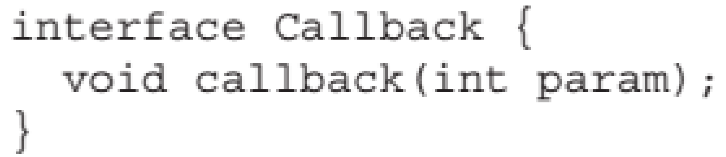
```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);

        type final-varname1 = value;
        type final-varname2 = value;
        //...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

Implementing Interfaces

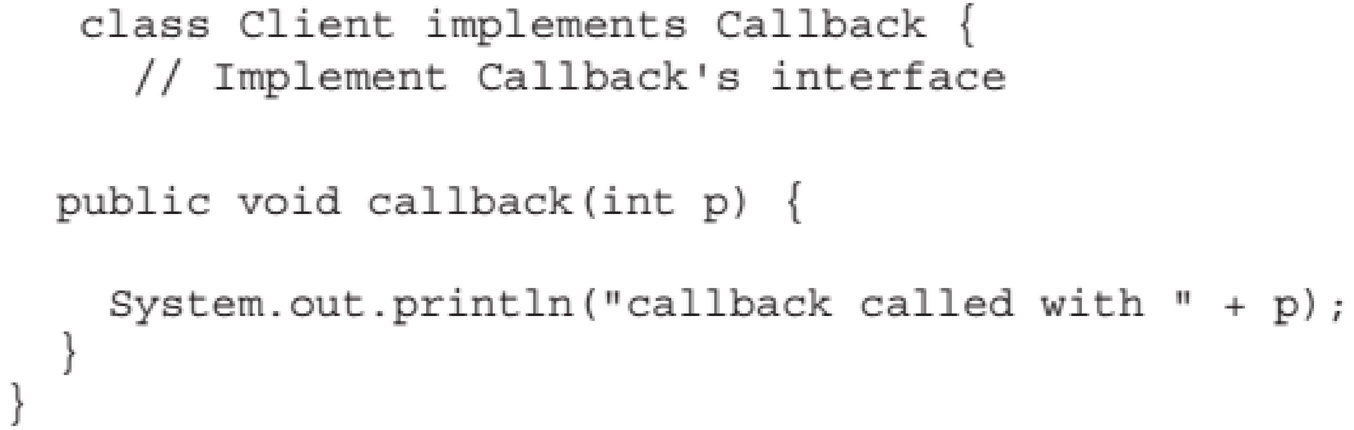
- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface.
- The general form of a class that includes the **implements** clause looks like this:

```
 class classname [extends superclass] [implements interface [,interface...]]
{
        // class-body
}
```

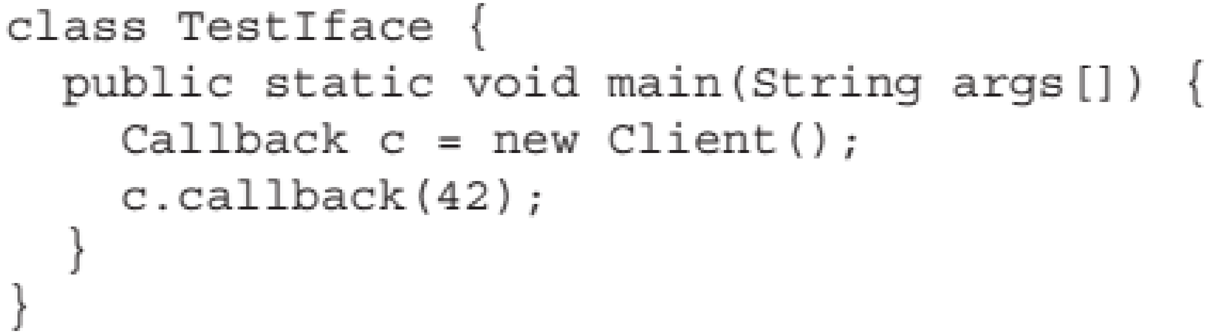
Example #3



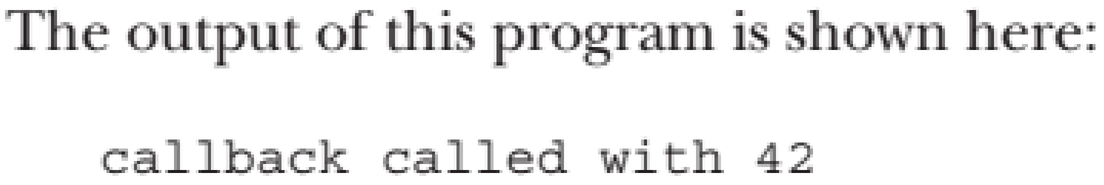
```
interface Callback {
  void callback(int param);
}
```



```
 class Client implements Callback {
   // Implement Callback's interface

  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```



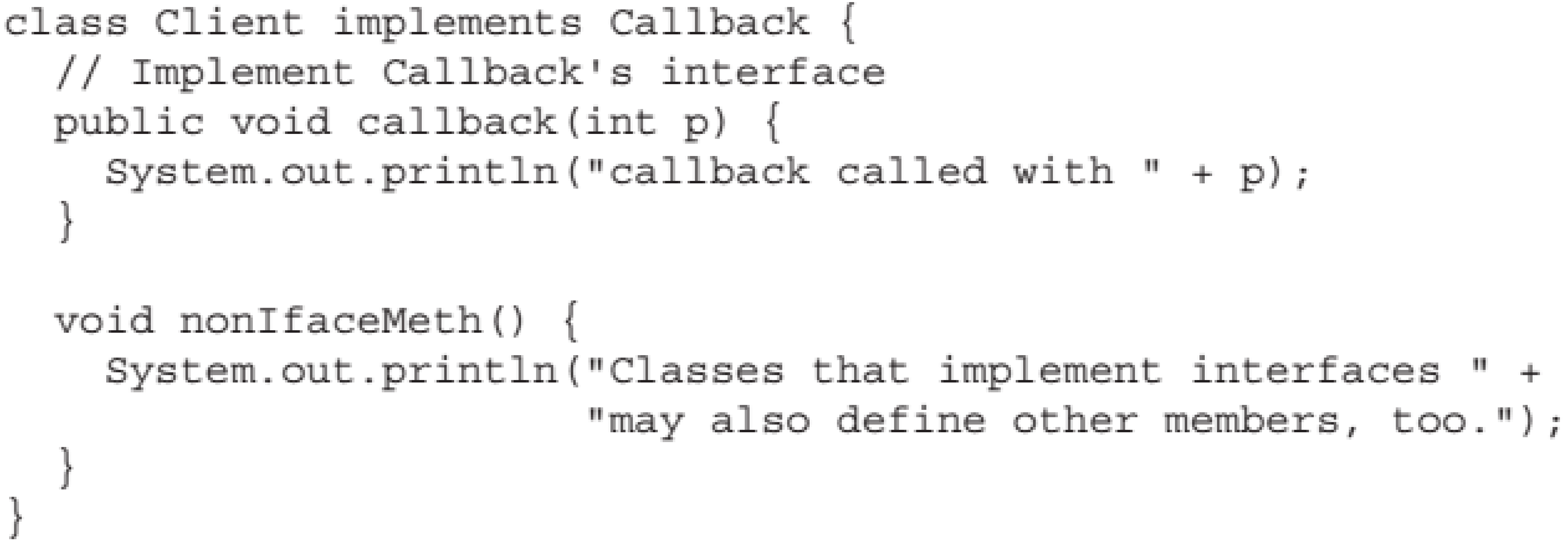
```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```



The output of this program is shown here:

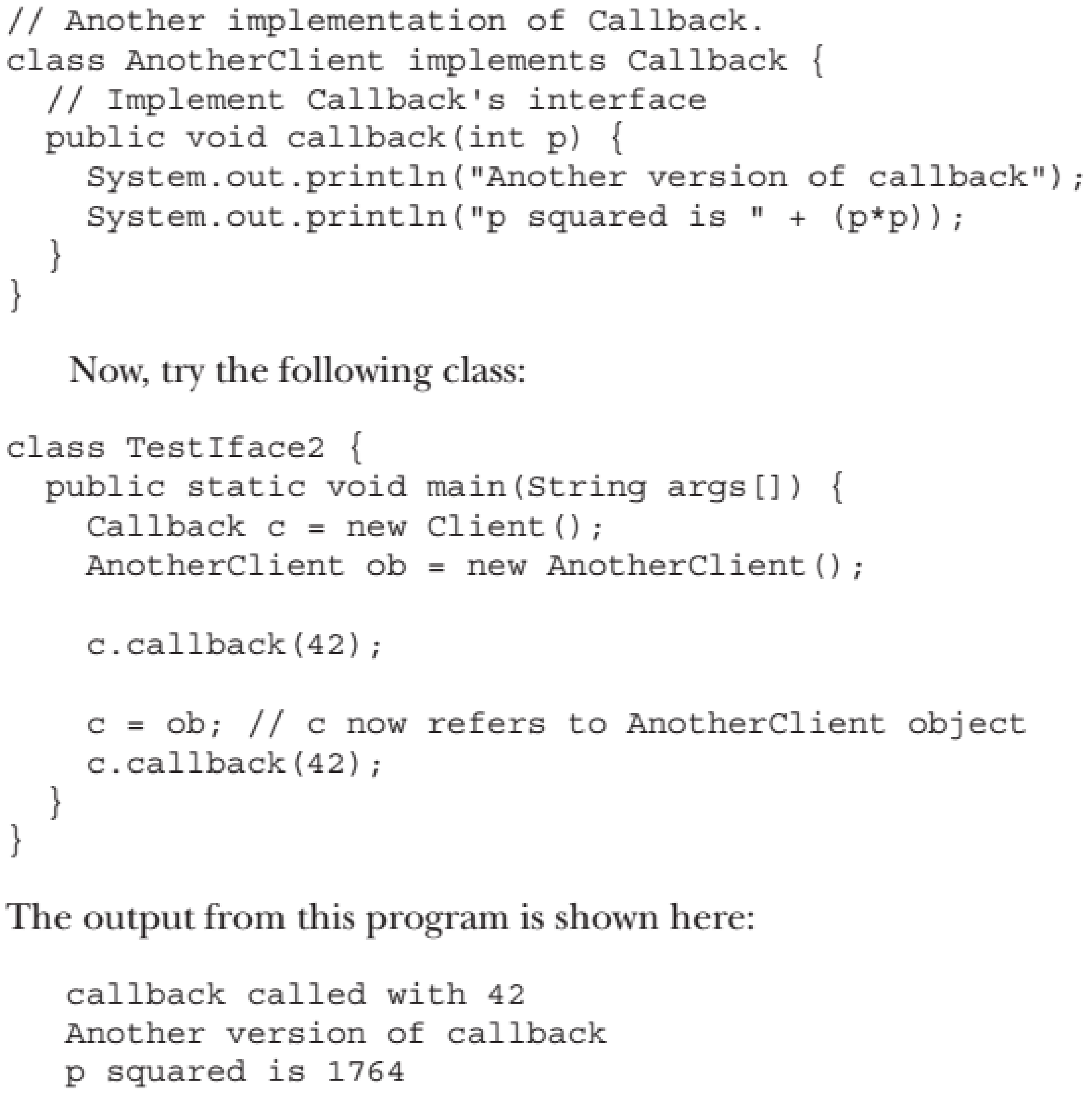
```
callback called with 42
```

Example #4



```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Example #5



```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
```

Now, try the following class:

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

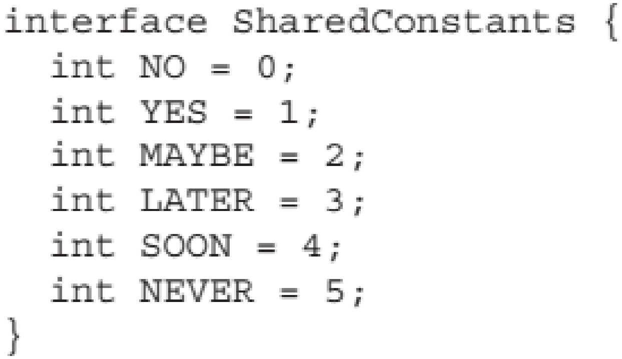
The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Nested Interfaces

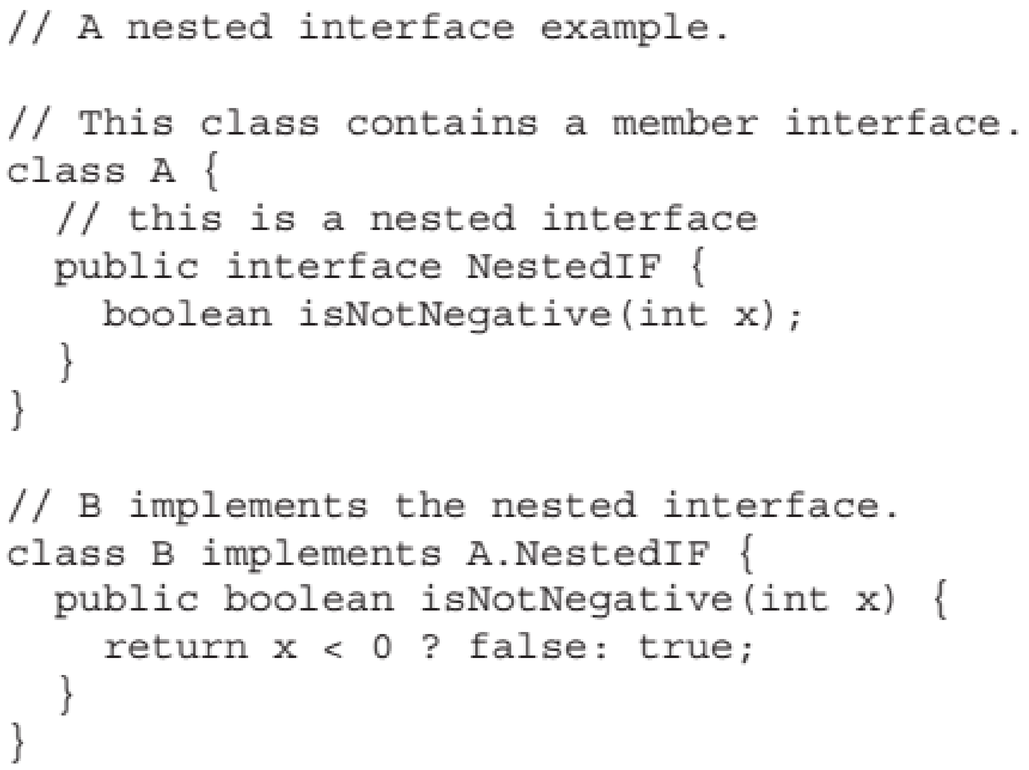
- Nested Interfaces
  - An interface can be declared a member of a class or another interface.
  - Such an interface is called a *member interface* or a *nested interface*.
  - A nested interface can be declared as **public**, **private**, or **protected**.
- Applying Interfaces best example is stack – Refer the example in Core java
- Variables in Interfaces
  - This is similar to using a header file in C/C++ to create a large number

    of **#defined** constants or **const** declarations

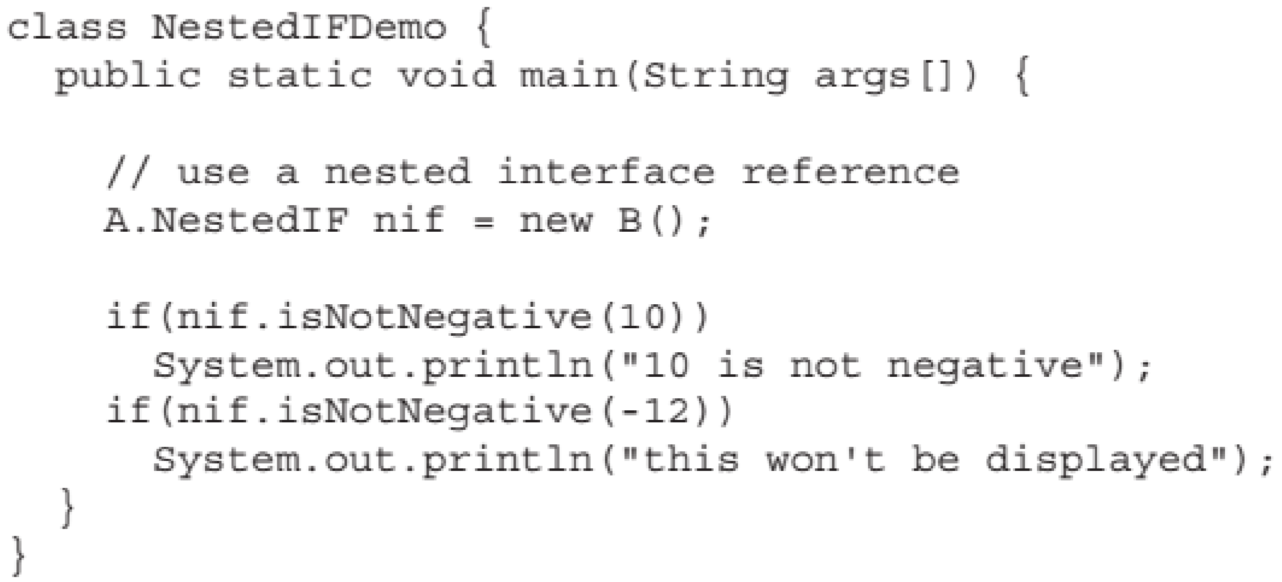


```
interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
```

Nested Interfaces



```
// A nested interface example.

// This class contains a member interface.
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}
```



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

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

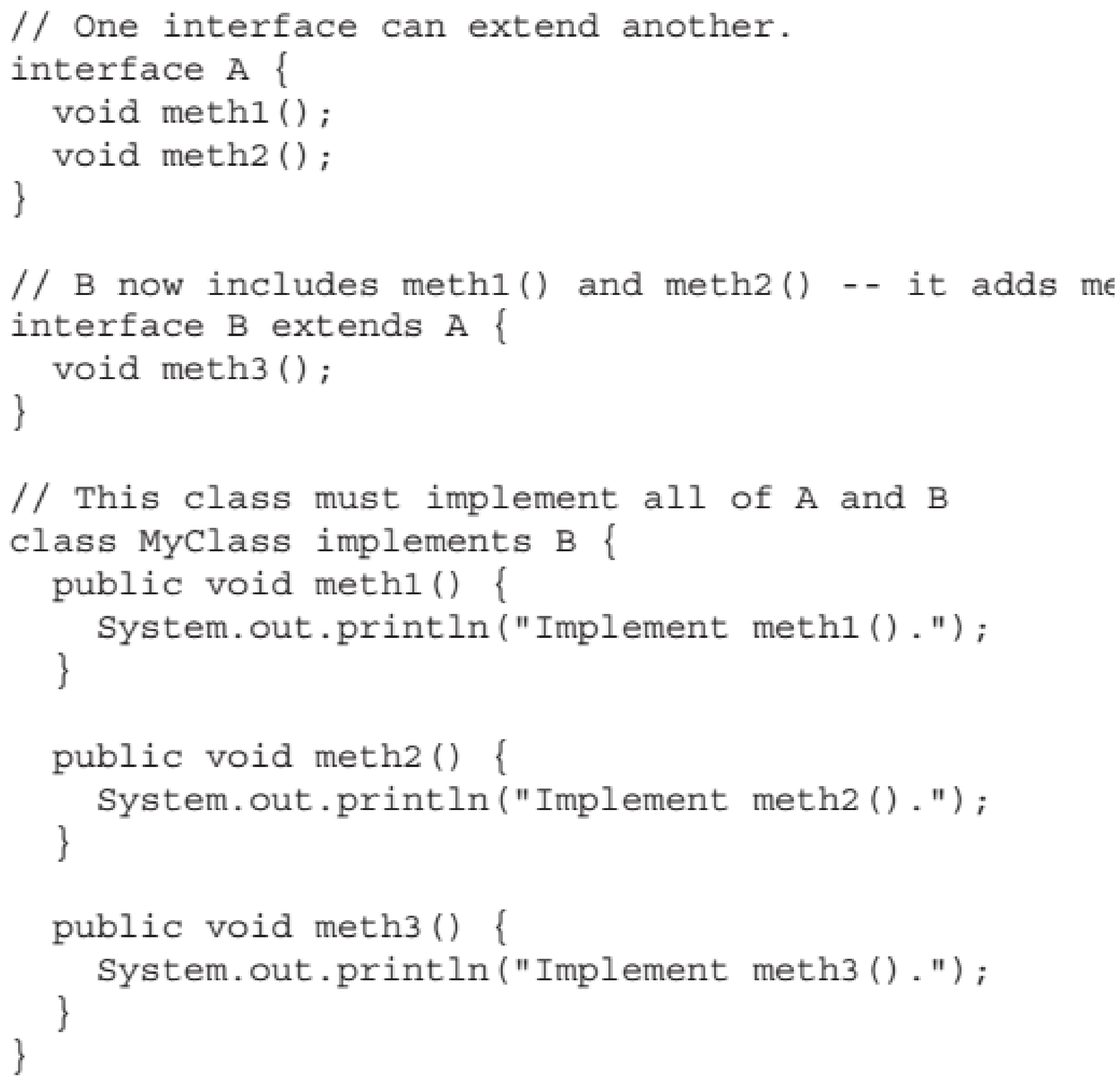
Example #6

Notice that A defines a member interface called NestedIF and that it is declared public.
Next, B implements the nested interface by specifying implements A.NestedIF

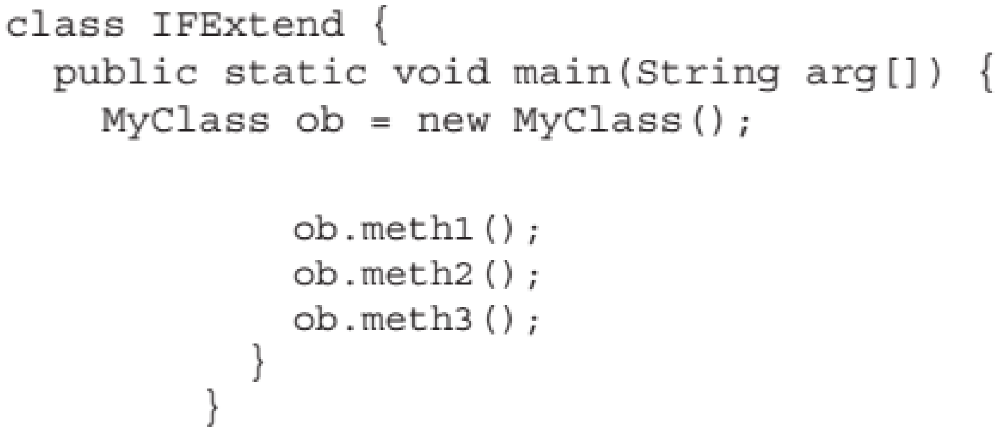
Example #7

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.



```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds me
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }
}
```



```
class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

            ob.meth1();
            ob.meth2();
            ob.meth3();
          }
        }
```