UNIT - IV

Packages & Interfaces

Packages & Interfaces

Packages: Defining a Package, Finding Packages and CLASSPATH, Access Protection, Importing Packages.

Interfaces: Defining an Interface, Implementing Interfaces, Nested Interfaces, Applying Interfaces, Variables in Interfaces, Interfaces Can Be Extended.

Packages and Interfaces

Packages:

- Defining a Package
- Finding Packages and CLASSPATH
- Access Protection
- Importing Packages

Interfaces:

- Defining an Interface
- Implementing Interfaces
- Nested Interfaces Applying Interfaces
- Variables in Interfaces
- Interfaces Can Be Extended

Packages

Basics:

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- **Eg:** A package allows you to create a class named **List,** which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- A Java's mechanism for partitioning the class name space into more manageable chunks.

Packages

Basics: ...

- The package is both a naming and a visibility control mechanism.
- One can define classes inside a package that are not accessible by code outside that package.
- Also define class members that are only exposed 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.

Packages

Basics: ...

The benefits of organising classes into packages are:

- The classes contained in the packages of other programs/applications can be reused.
- In packages classes can be unique compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
- Classes in packages can be hidden if we don’t want other packages to access them.
- Also provide a way for separating “design” from coding.
- ***Packages enable grouping of functionally related classes.***

Packages

Basics: ...

The Java Foundation Packages

- Java provides a large number of classes groped into different packages based on their functionality.

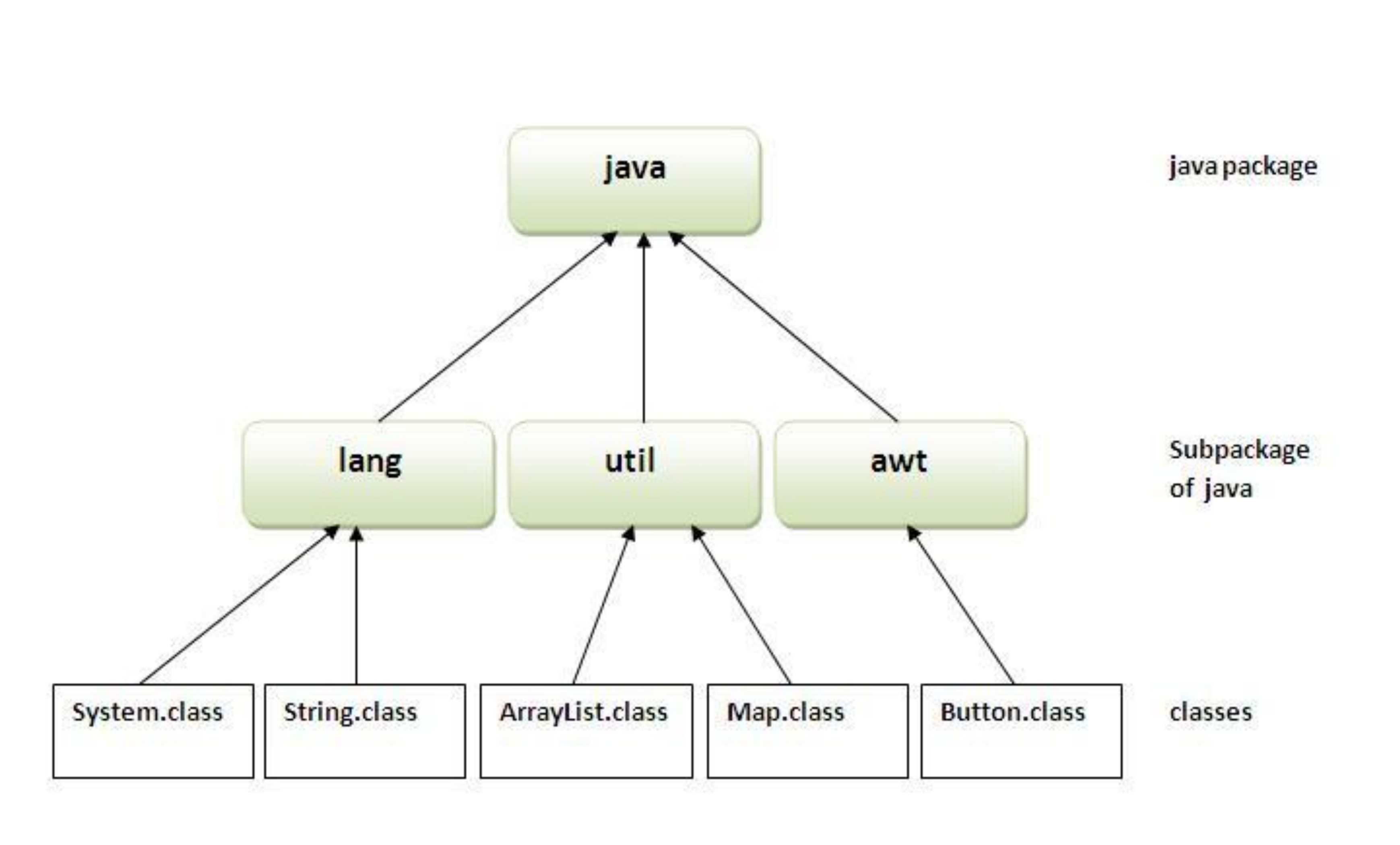
The six foundation Java packages are:

- ***java.lang:*** Classes for primitive types, strings, math functions, threads, and exception
- ***java.util:*** Classes such as vectors, hash tables, date etc.
- ***java.io:*** Stream classes for I/O
- ***java.awt:*** Classes for implementing GUI – windows, buttons, menus etc.
- ***java.net:*** Classes for networking
- ***java.applet:*** Classes for creating and implementing applets

Packages

Basics: ...

The Java Foundation Packages



Packages

Defining a Package:

- Include a **package** command as the first statement in a Java source file. 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.
- General Form: **package *pkg;*** **Eg:** package MyPackage;

Packages

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.

Packages

Defining a Package: ...

- 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 **package *pkg1[.pkg2[.pkg3]];***
- A package hierarchy must be reflected in the file system of your Java development system.
- **Eg:** A package declared as **package java.awt.image;** needs to be stored in **java\awt\image** in a Windows environment.
- Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Packages

Finding Packages and CLASSPATH:

- Packages are mirrored by directories. This raises an important question: ***How does the Java run-time system know where to look for packages that you create?***

- The answer has three parts.

✓First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

✓Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

✓Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

Packages

Finding Packages and CLASSPATH: ...

- Eg: ***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*

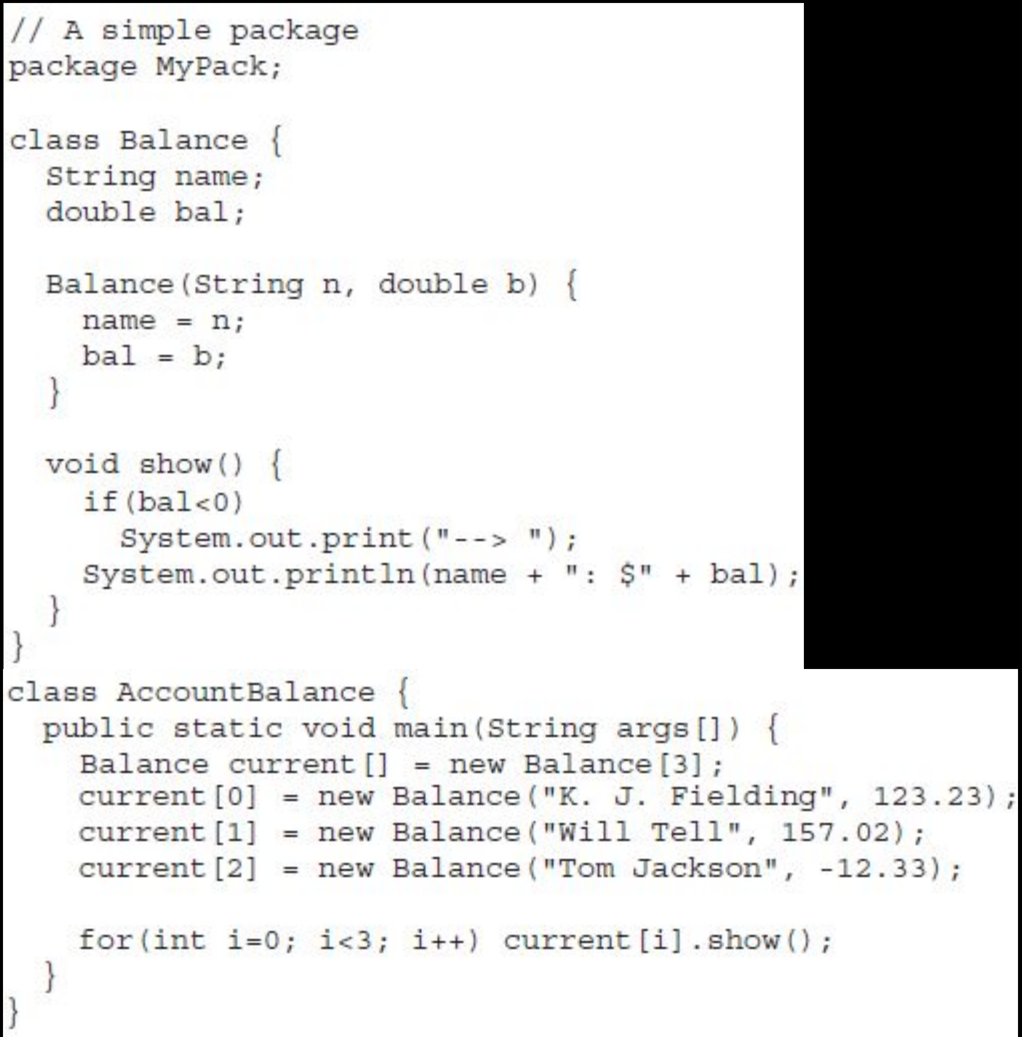
✓the **-classpath** option must specify the path to **MyPack** when the program is run via **java.**

- When the second two options are used, the class path *must not include **MyPack***, itself. It must simply specify the *path to* **MyPack.** For example, in a Windows environment, if the path to **MyPack** is C:\MyPrograms\Java\MyPack Then the class path to **MyPack** is C:\MyPrograms\Java\MyPack

Packages

Finding Packages and CLASSPATH: ...

- A Short Package Example



```
// 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();
  }
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack.** Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

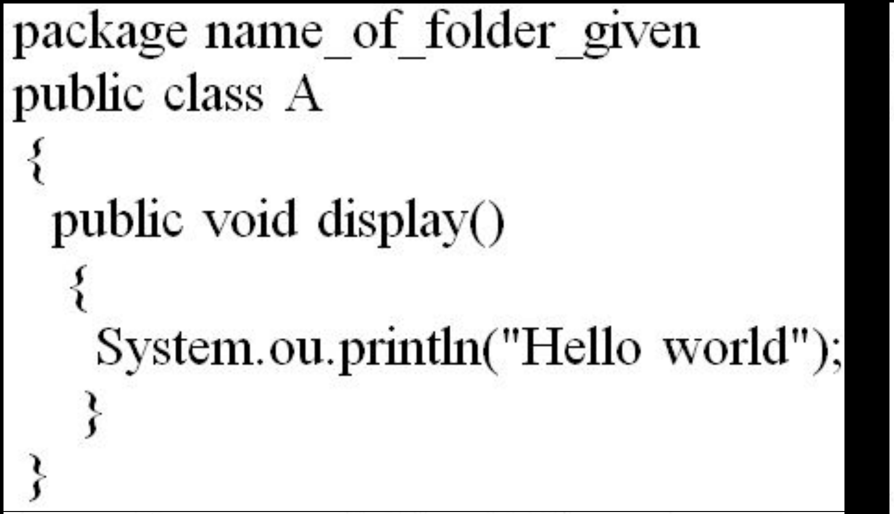
Remember, you will need to be in the directory above **MyPack** when you execute this command.

AccountBalance is now part of the package **MyPack.** This means that it cannot be executed by itself.

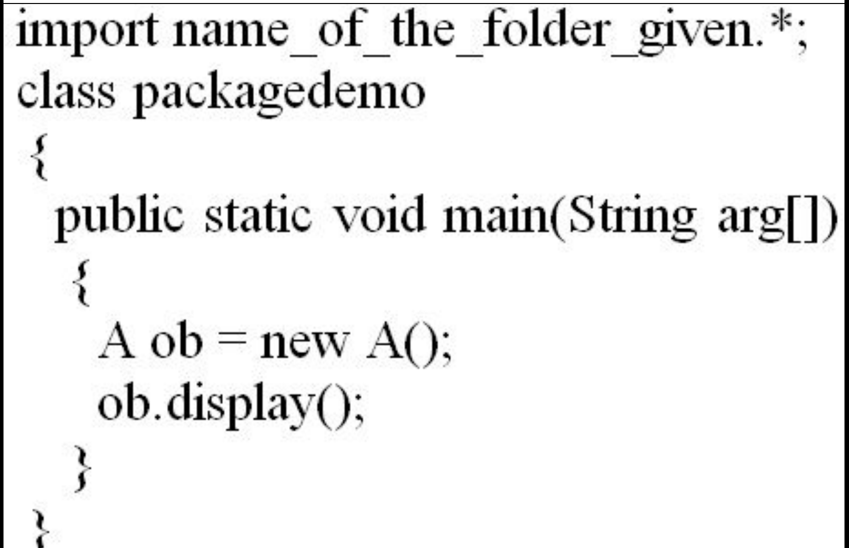
Packages

Finding Packages and CLASSPATH: ...

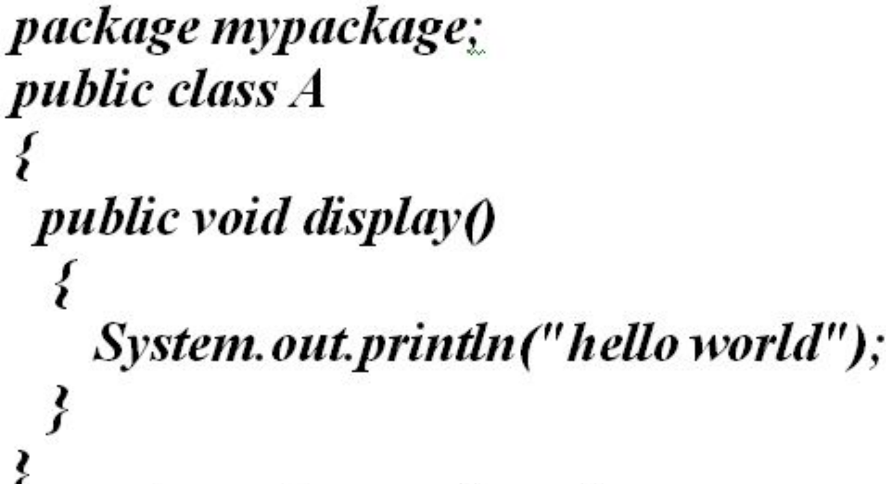
- A Short Package Example ...



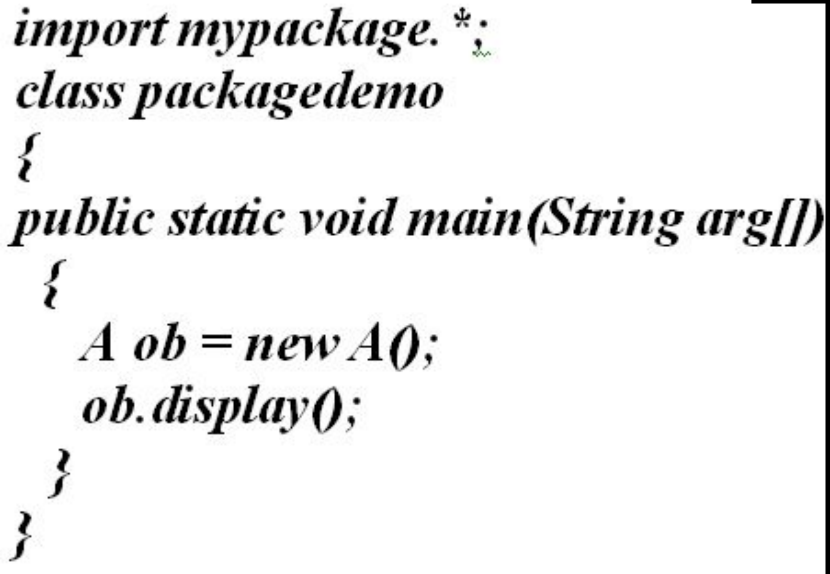
```
package name_of_folder_given
public class A
 {
   public void display()
    {
     System.ou.println("Hello world");
    }
 }
```



```
import name_of_the_folder_given.*;
class packagedemo
 {
   public static void main(String arg[])
    {
      A ob = new A();
      ob.display();
    }
 }
```



```
package mypackage;
public class A
{
 public void display()
  {
     System.out.println("hello world");
  }
}
```



```
import mypackage.*;
class packagedemo
{
public static void main(String arg[])
  {
    A ob = new A();
    ob.display();
  }
}
```

Packages

Access Protection:

- Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- 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.

Packages

Access Protection: ...

- 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 specifiers, **private, public,** and **protected,** provide a variety of ways to produce the many levels of access required by these categories.

Packages

Access Protection: ...

- While Java’s access control mechanism may seem complicated, we can simplify it as follows:

- Anything declared **public** can be accessed from anywhere.

- Anything declared **private** cannot be seen outside of its class.

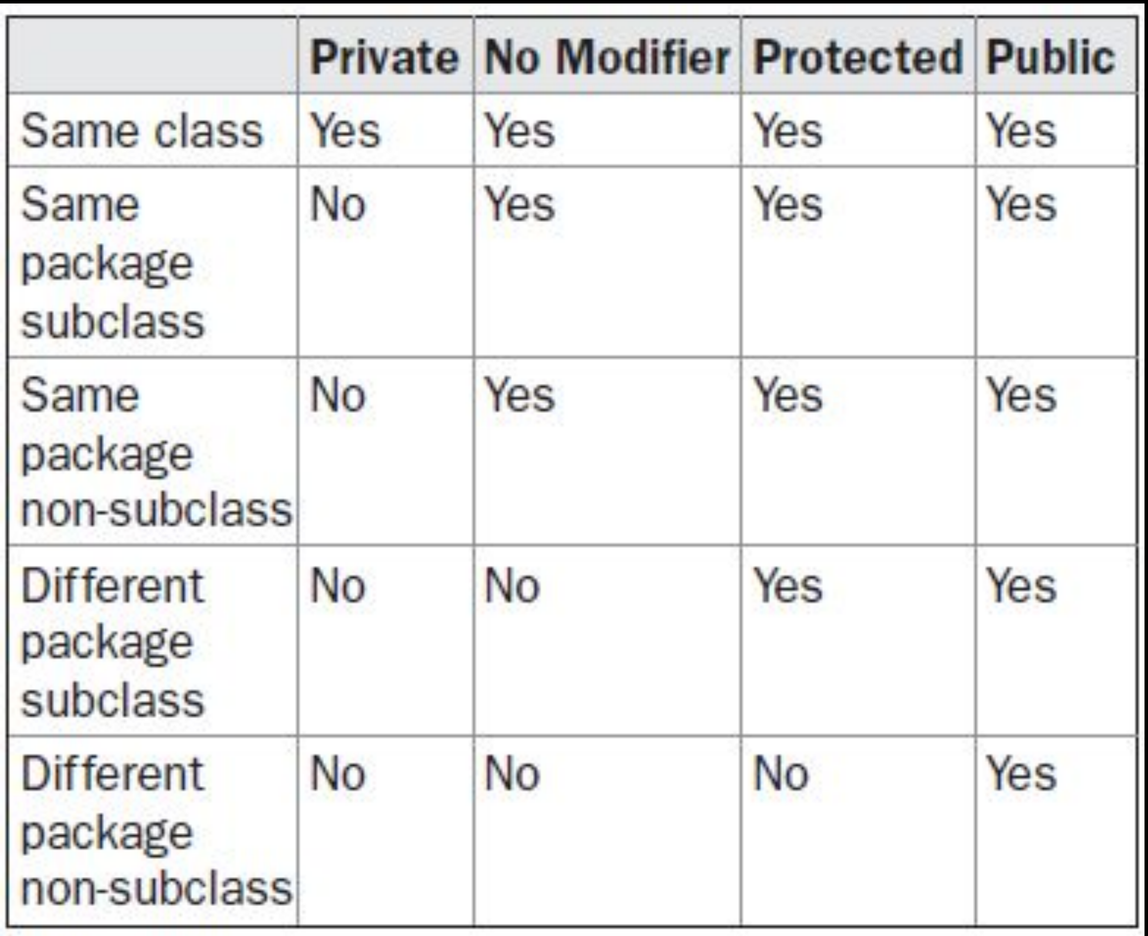
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.

- 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**.

Packages

Access Protection: ...

- Class Member Access - ***Applies only to members of classes.***



	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

A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code.

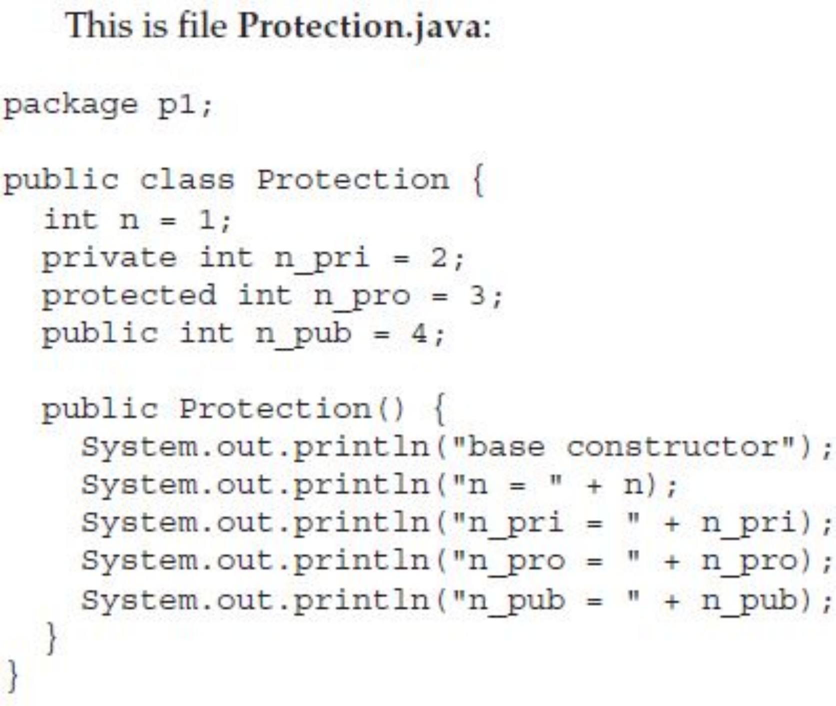
If a class has default access, then it can only be accessed by other code within its same package.

When a class is **public**, it must be the only public class declared in the file, and the file must have the same name as the class.

Packages

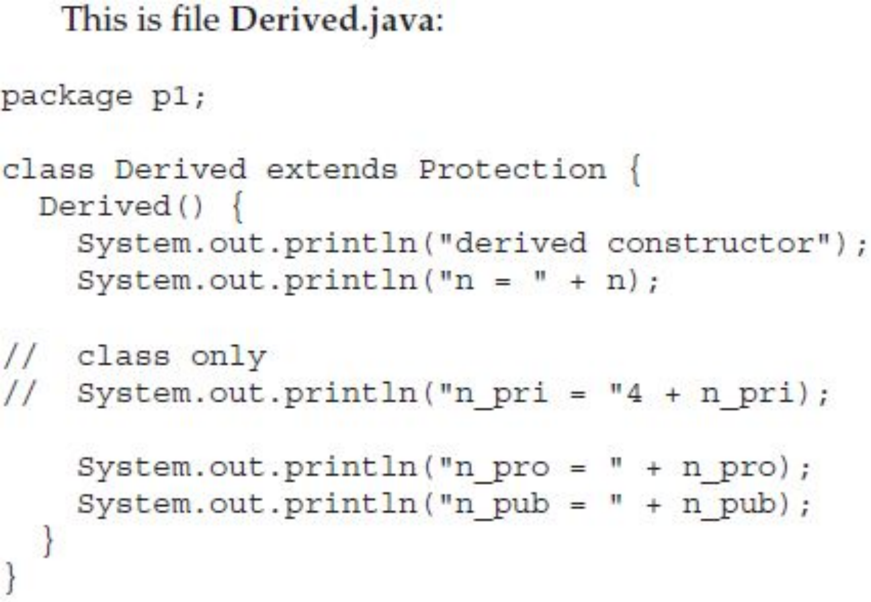
Access Protection: ...

- *An Access Example*



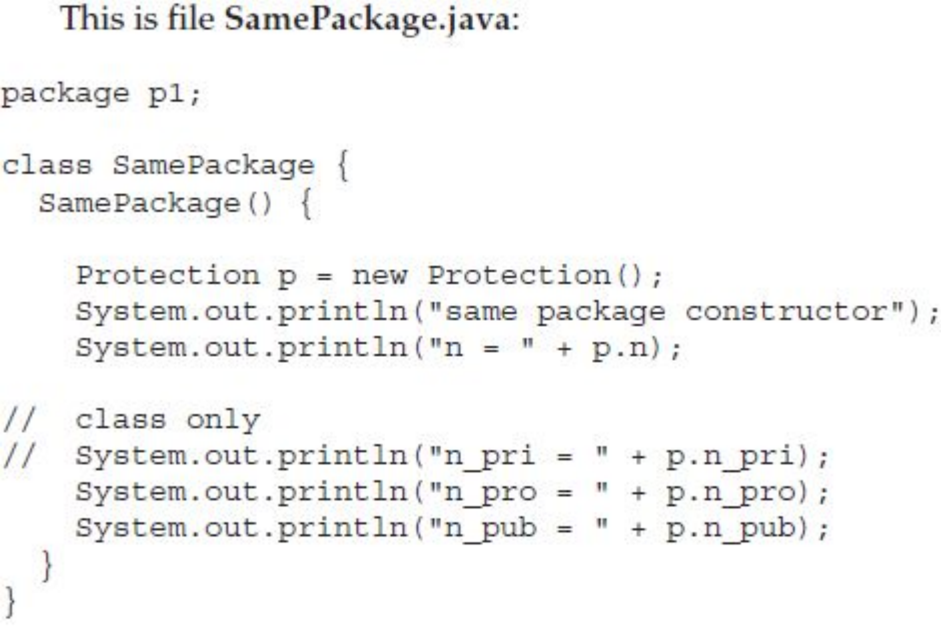
This is file **Protection.java**:

```
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);
  }
}
```



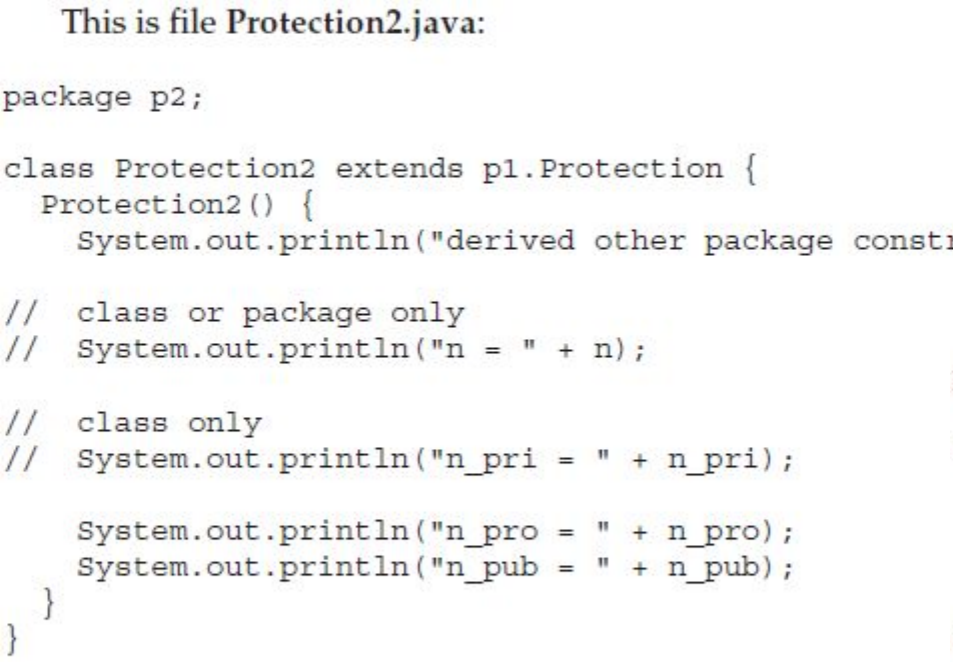
This is file **SamePackage.java**:

```
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);
  }
}
```

Packages

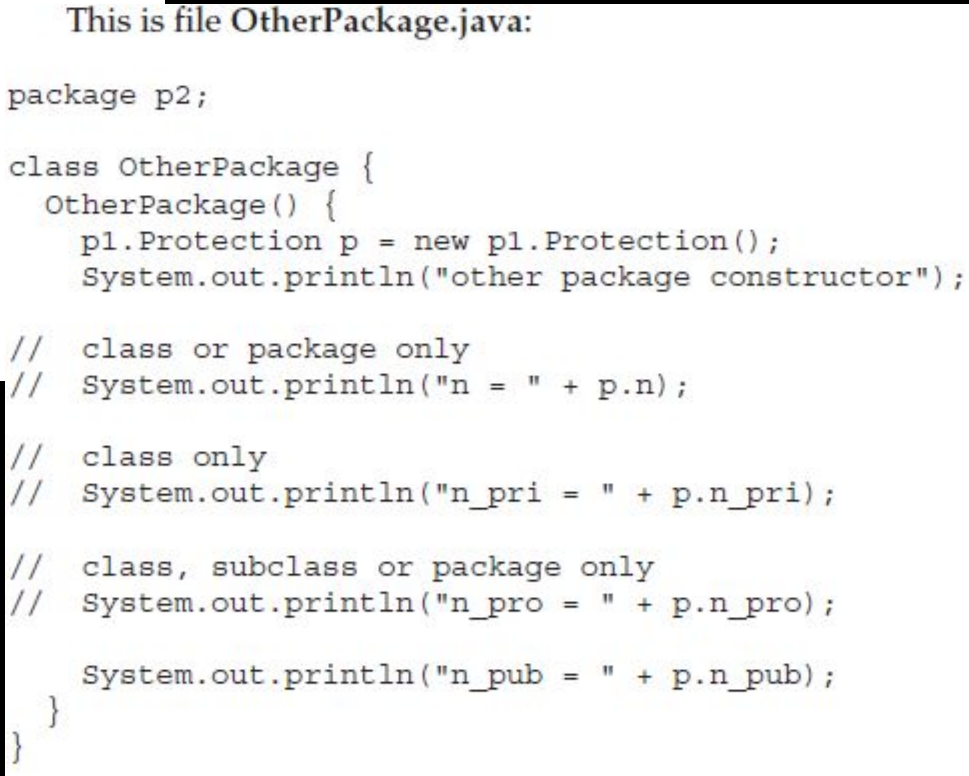
Access Protection: ...

- *An Access Example ...*



This is file Protection2.java:

```
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);
  }
}
```



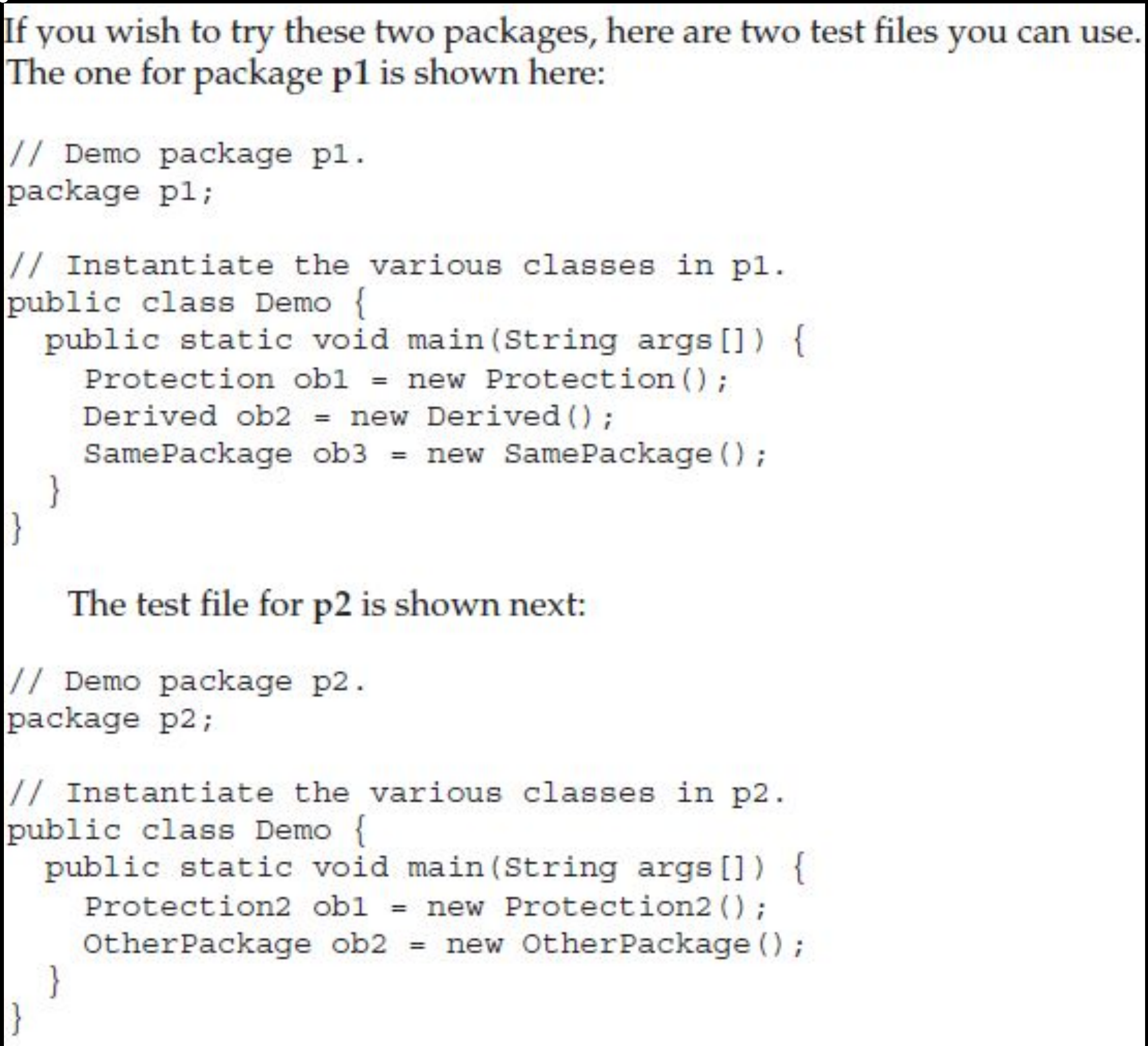
This is file OtherPackage.java:

```
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);
  }
}
```

Packages

Access Protection: ...

- *An Access Example ...*



If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// 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();
  }
}
```

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();
  }
}
```

Packages

Importing Packages:

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
- For this reason, 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.

Packages

Importing Packages: ...

- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. The general form of the import statement:

import pkg1[.pkg2].(classname|*);

- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.).

Packages

Importing Packages: ...

- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

- ***CAUTION*** *The star form may increase compilation time-especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.*

Packages

Importing Packages: ...

- 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**
- Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs: ***import java.lang.*;***
- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

Packages

Importing Packages: ...

- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name,* which includes its full package hierarchy.
- For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

- The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

- In this version, **Date** is fully-qualified.

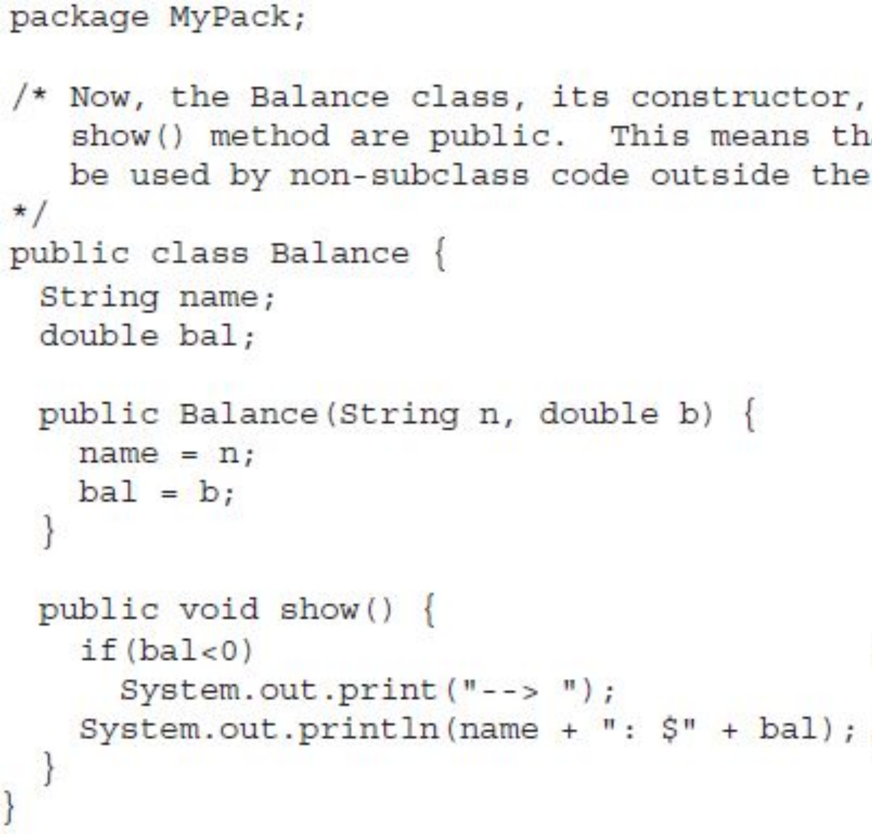
Packages

Importing Packages: ...

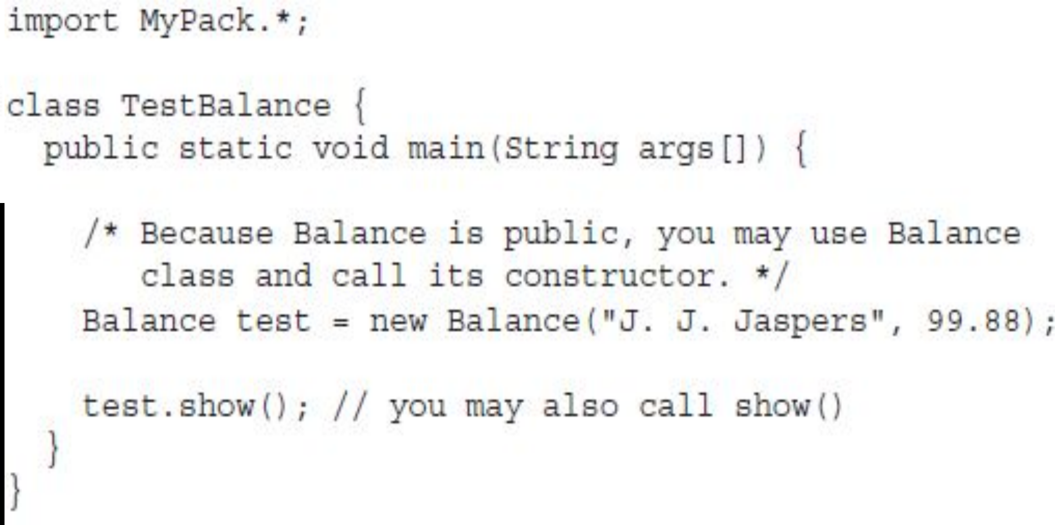
- When a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.
- For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file.

Packages

Importing Packages: ...



```java
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public.  This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

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

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



```java
import MyPack.*;

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

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

Packages

Additional Resources:

Packages-1

Packages-2

Packages-3

Interfaces

Basics:

- Through the use of the **interface** keyword, Java allows you to fully abstract the interface from its implementation.
- Using **interface**, you can specify a set of methods that can be implemented by one or more classes. The **interface,** itself, does not actually define any implementation.
- Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Interfaces

Basics: ...

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation. Also 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 their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface.** Also, one class can implement any number of interfaces.

Interfaces

Basics: ...

- To implement an **interface**, a class must create the complete set of methods defined 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.
- Interfaces are designed to support dynamic method resolution at run time.
- ***NOTE*** *Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.*

Interfaces

Basics: ...

Why use Java interface? *There are mainly three reasons:*

It is used to achieve fully abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

- The java compiler adds **public** and **abstract** keywords before the interface method and **public, static** and **final** keywords before data members.

Interfaces

Basics: ...

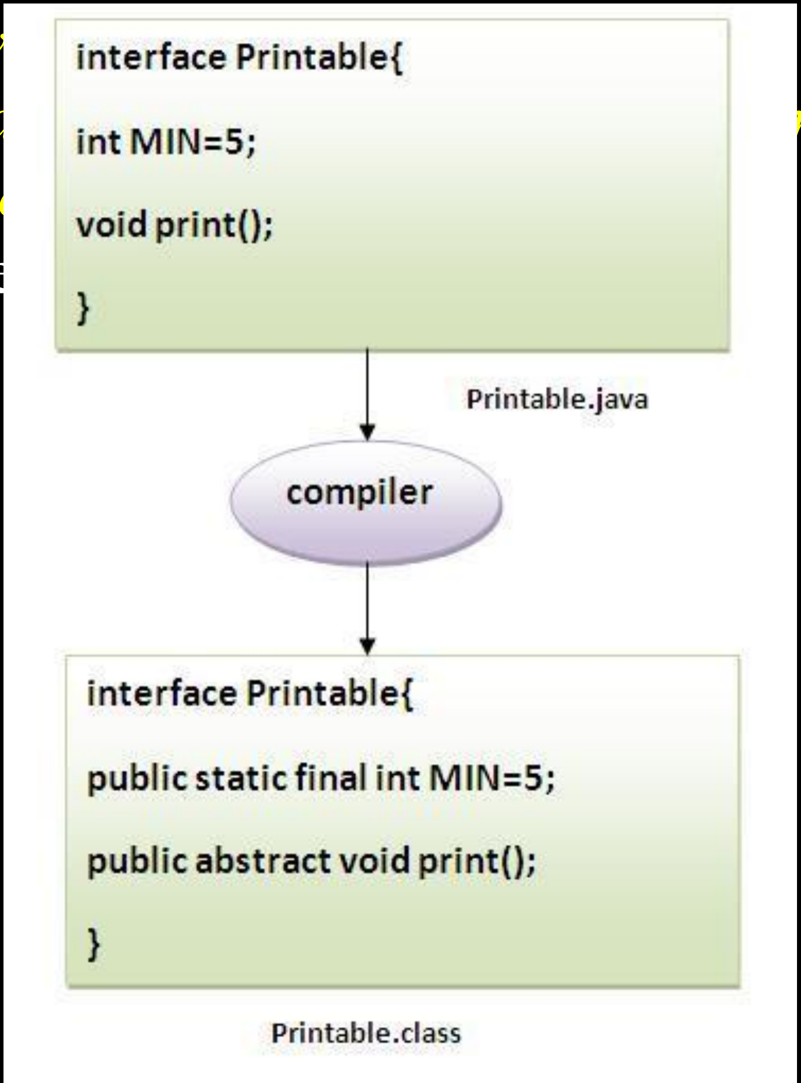
Why use Java interface? *There are mainly three reasons:*

It is used to achiev

By interface, we ca *multiple inheritance.*

It can be used to a

- The java compile keywords before the interface method keywords before data members.



```
interface Printable{
int MIN=5;
void print();
}
```

Printable.java

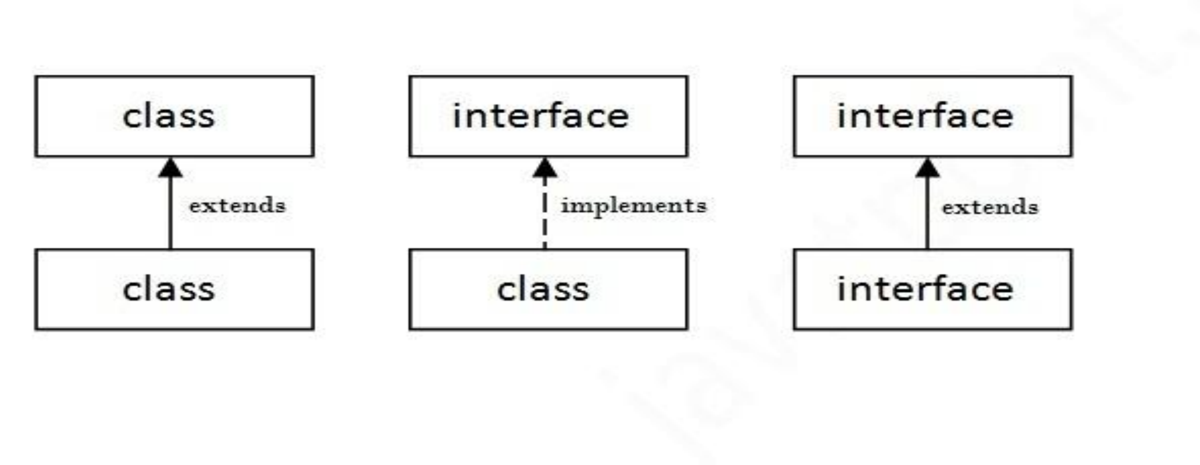
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

Printable.class

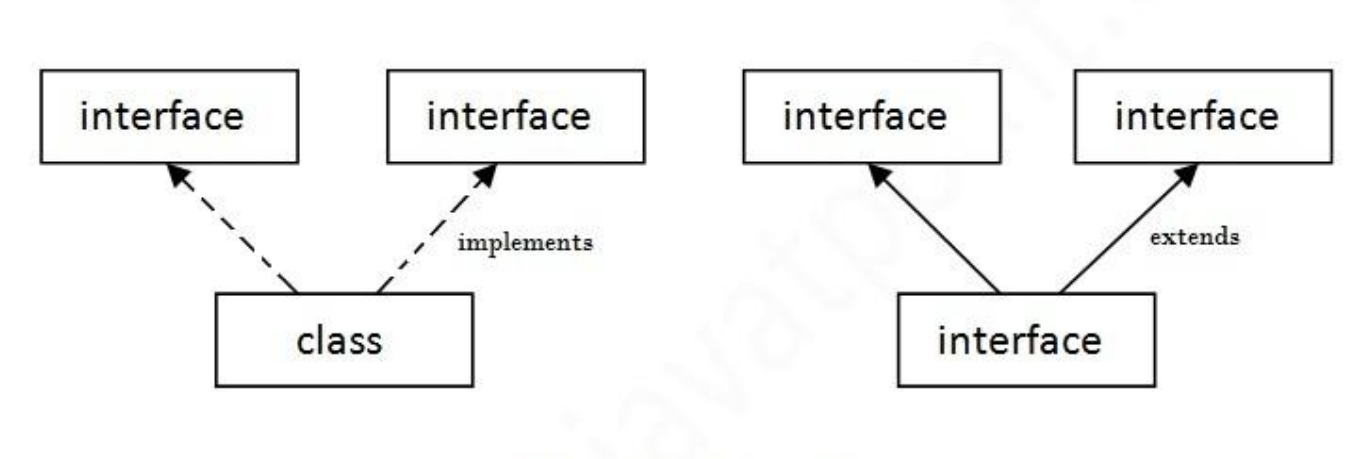
Interfaces

Basics: ...

- Understanding relationship between classes and interfaces



- Multiple inheritance in Java by interface



Multiple Inheritance in Java

Interfaces

Defining an Interface:

- An interface is defined much like a class. The General form:

```
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;
}
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public,** the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.

Interfaces

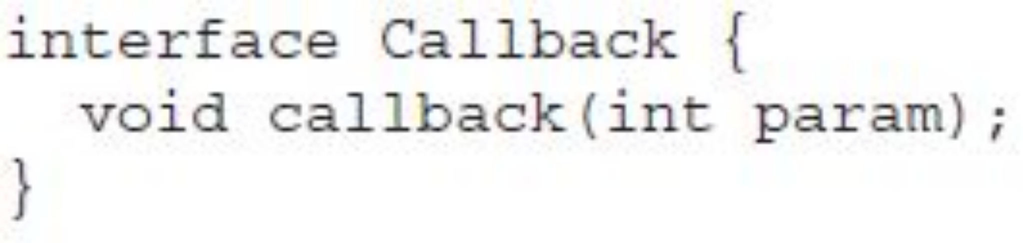
Defining an Interface: ...

- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static,** meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public.**

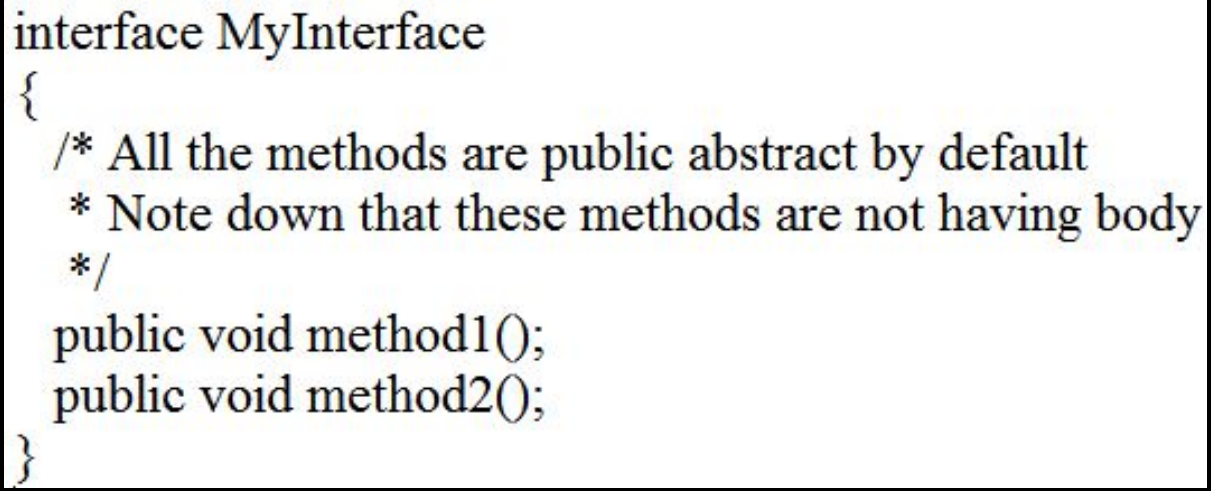
Interfaces

Defining an Interface: ...

- Examples:



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

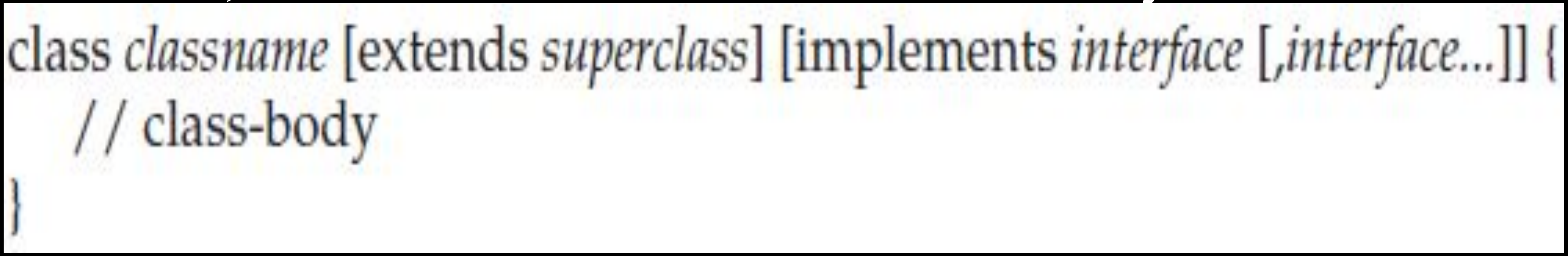


```
interface MyInterface
{
   /* All the methods are public abstract by default
    * Note down that these methods are not having body
    */
   public void method1();
   public void method2();
}
```

Interfaces

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 defined by the interface.

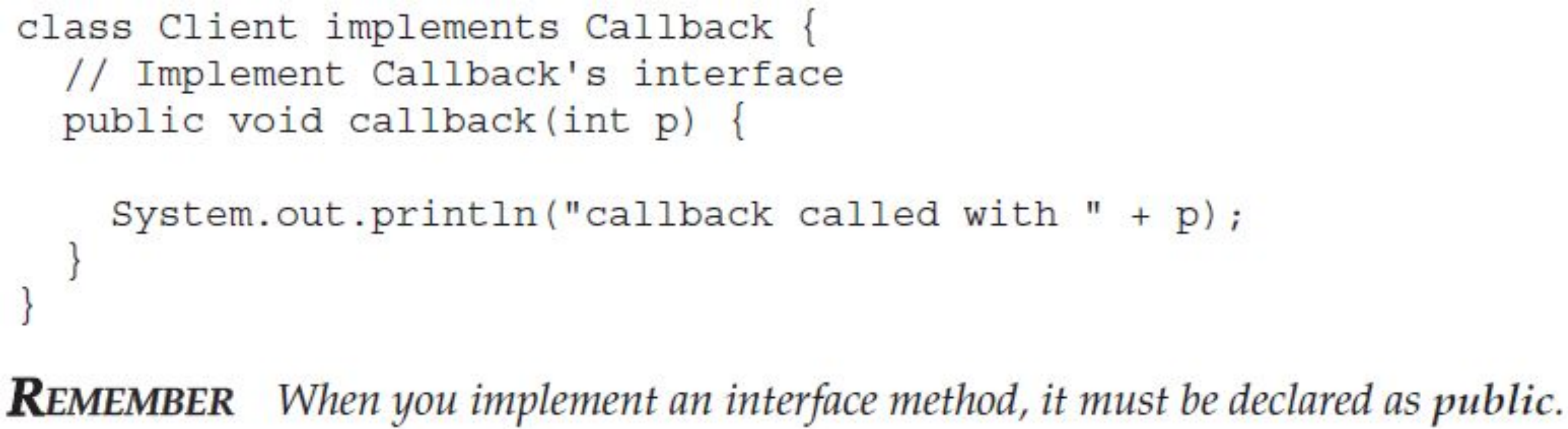


```
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Interfaces

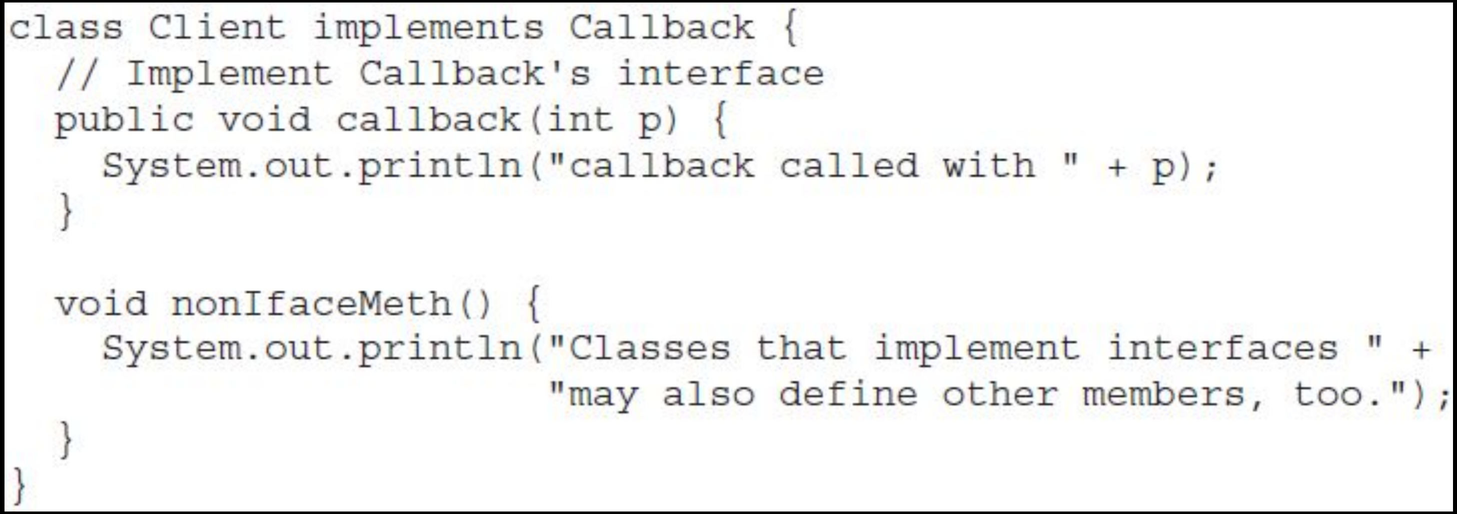
Implementing Interfaces: ...



```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {

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

REMEMBER *When you implement an interface method, it must be declared as public.*

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

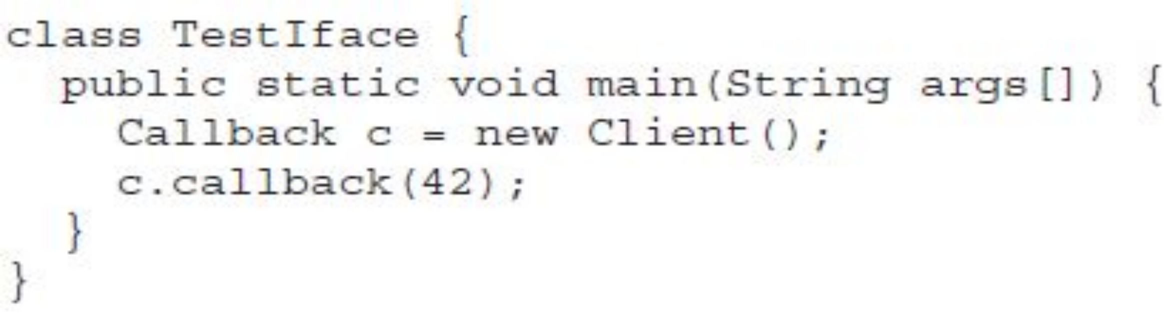


```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

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

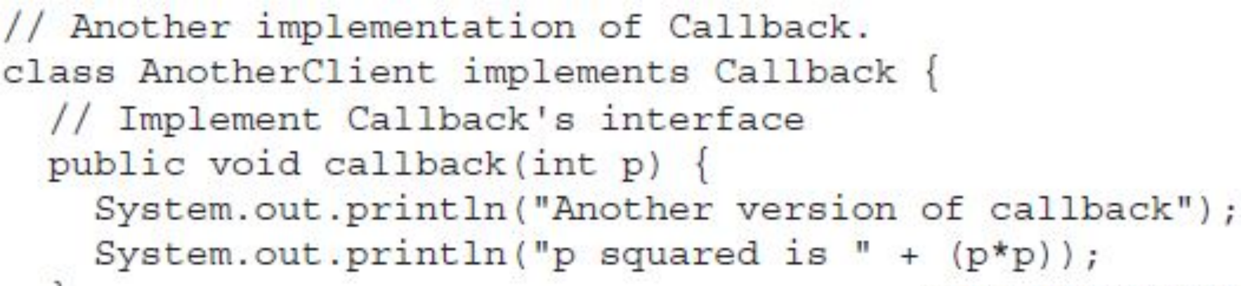
Interfaces

Implementing Interfaces: ...

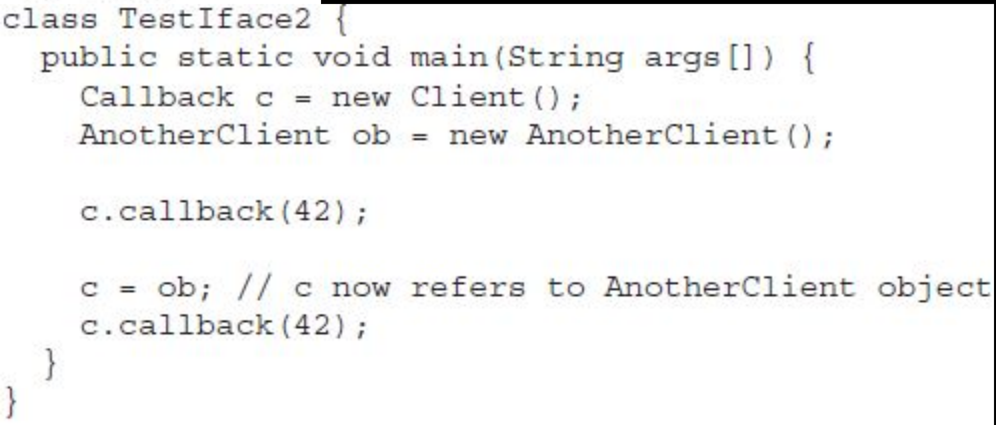
- Accessing Implementations Through Interface References



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



```
// 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));
  }
}
```

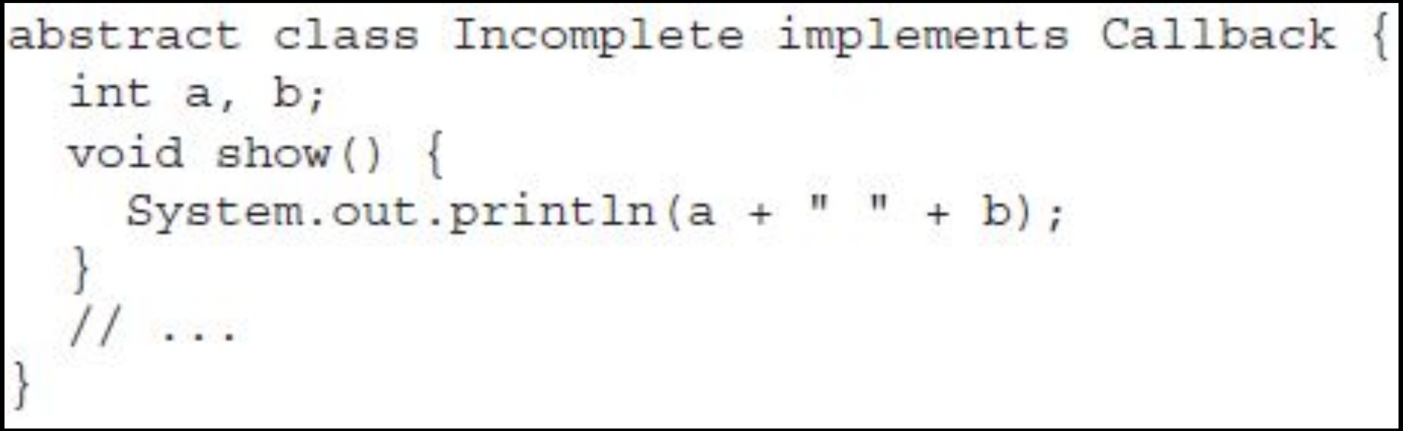


```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

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

Interfaces

Implementing Interfaces: ...

- **Partial Implementations**
- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract.**



```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

- Here, the class **Incomplete** does not implement **callback( )** and must be declared as **abstract.**
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

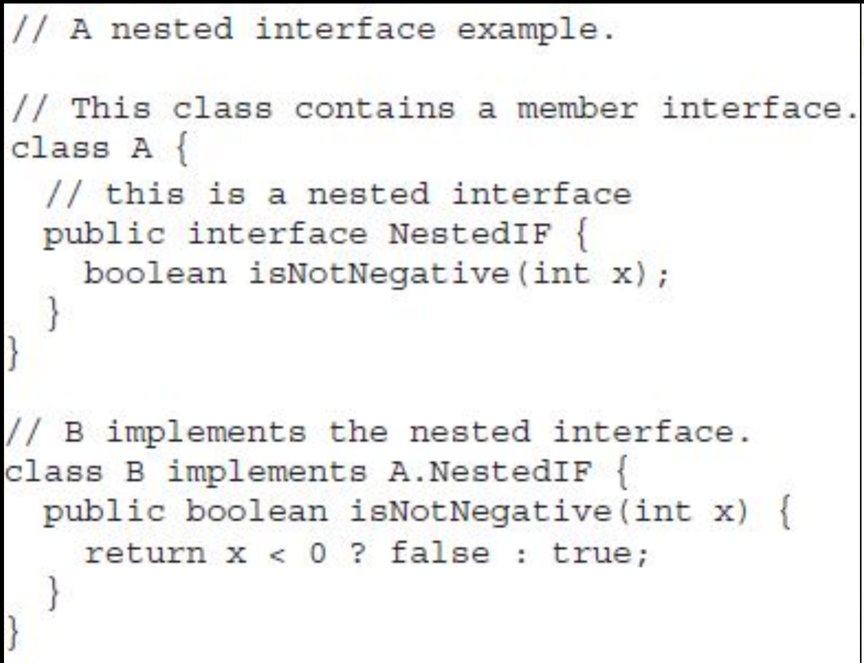
Interfaces

Nested Interfaces:

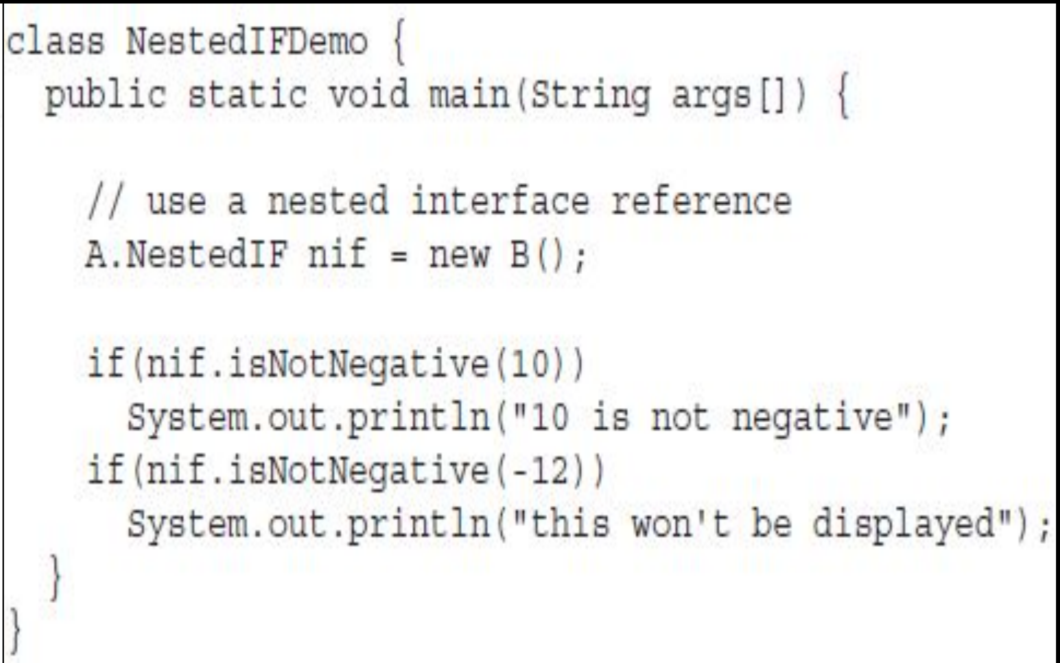
- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or *a nested interface.*
- A nested interface can be declared as **public, private, or protected.** This differs from a top-level interface, which must either be declared as **public** or use the **default** access level.
- When a **nested interface** is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Interfaces

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;
  }
}
```



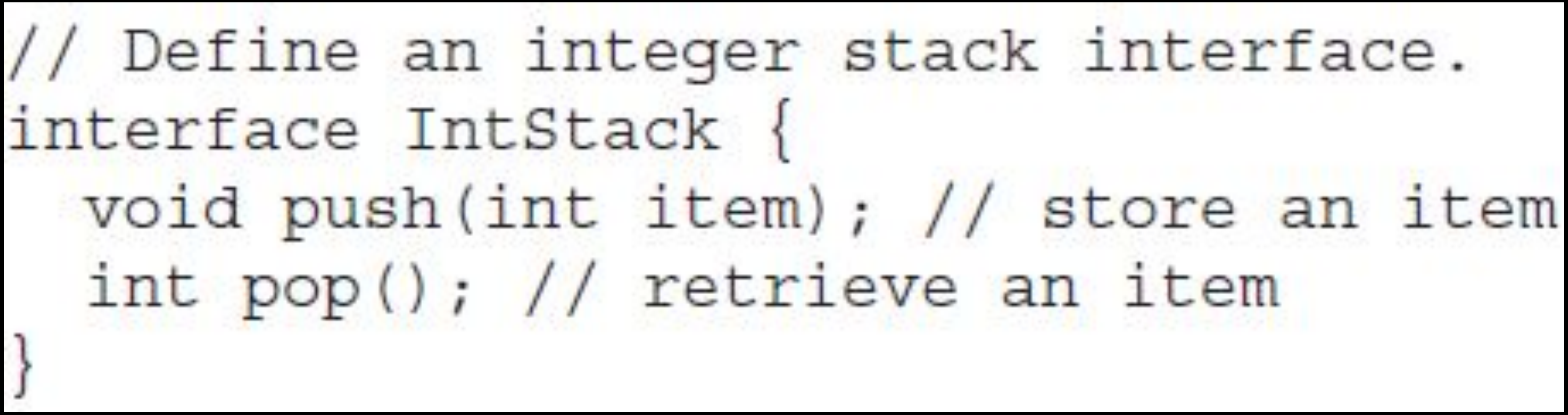
```
class NestedIFDemo {
  public static void main(String args[]) {

    // 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");
  }
}
```

- Notice that the name is fully qualified by the enclosing class’ name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF,** this is legal.

Interfaces

Applying Interfaces:

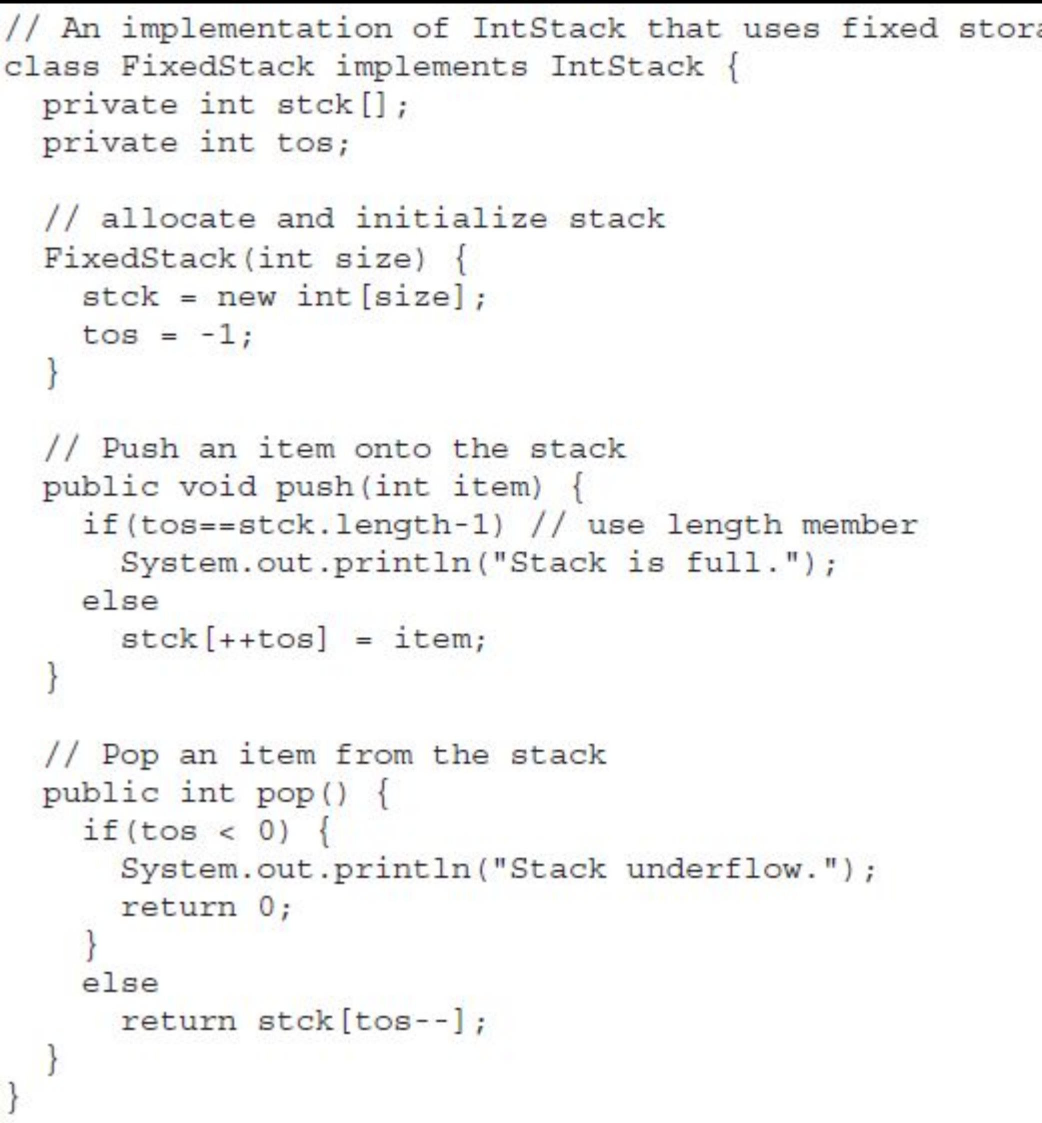
- The interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics.



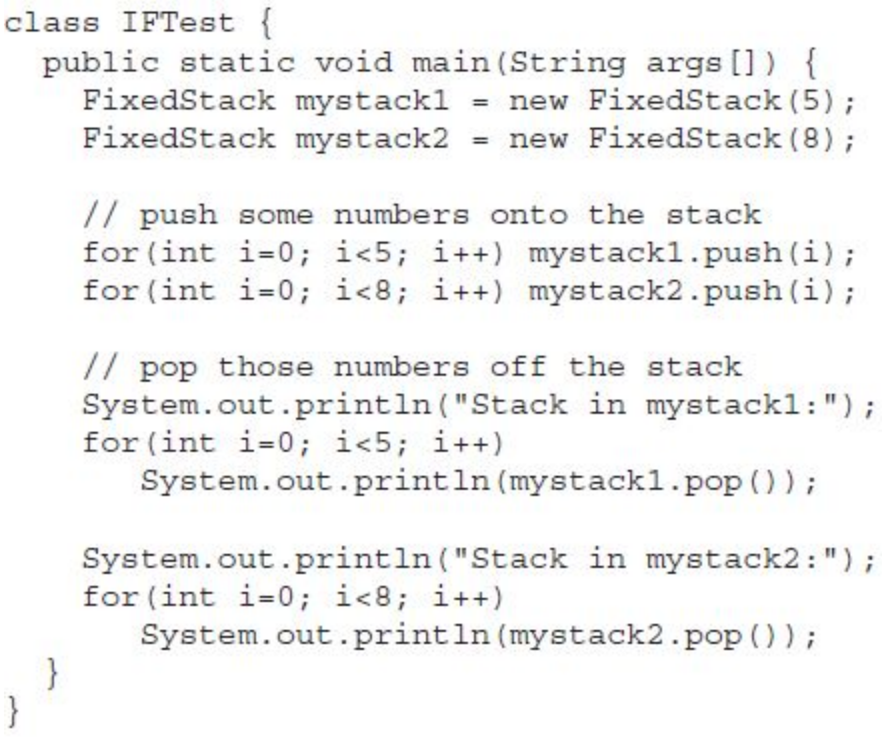
```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

Interfaces

Applying Interfaces: ...



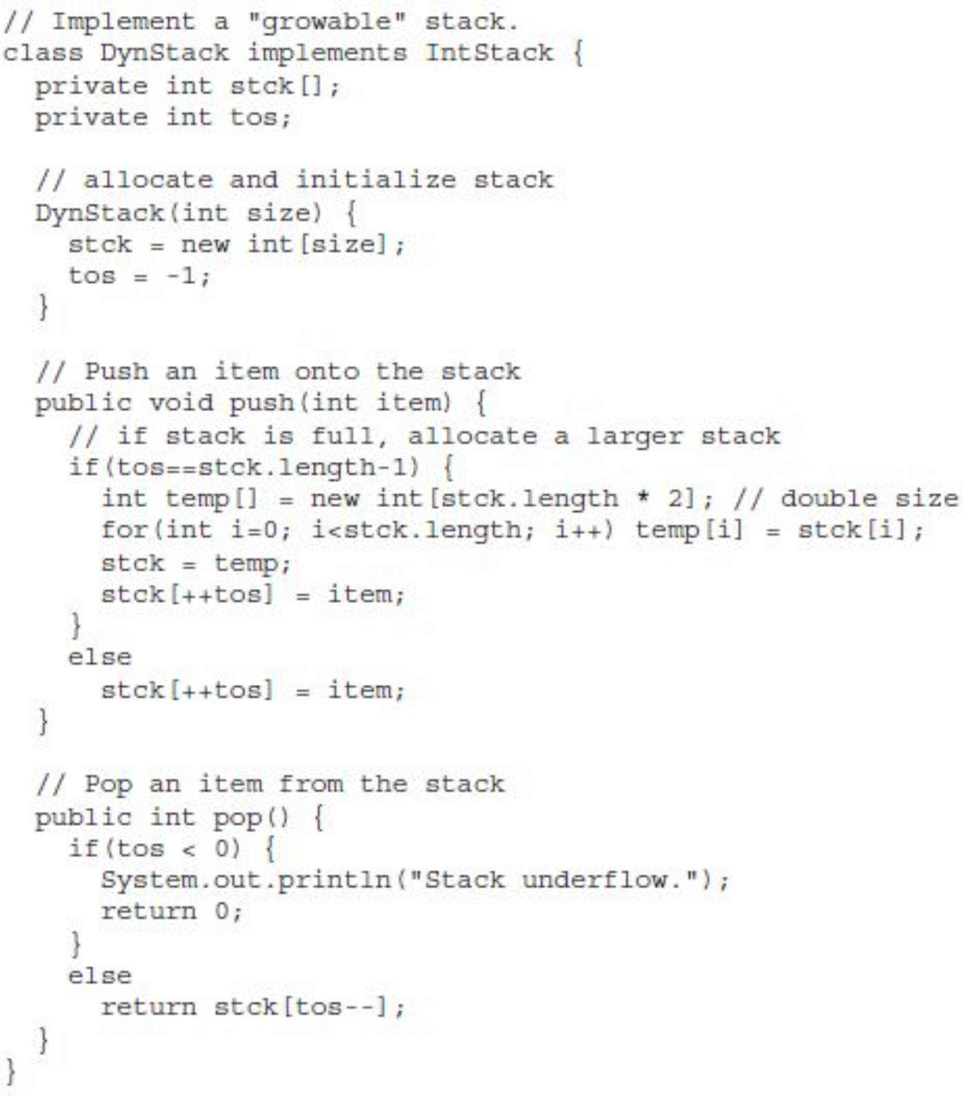
```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```



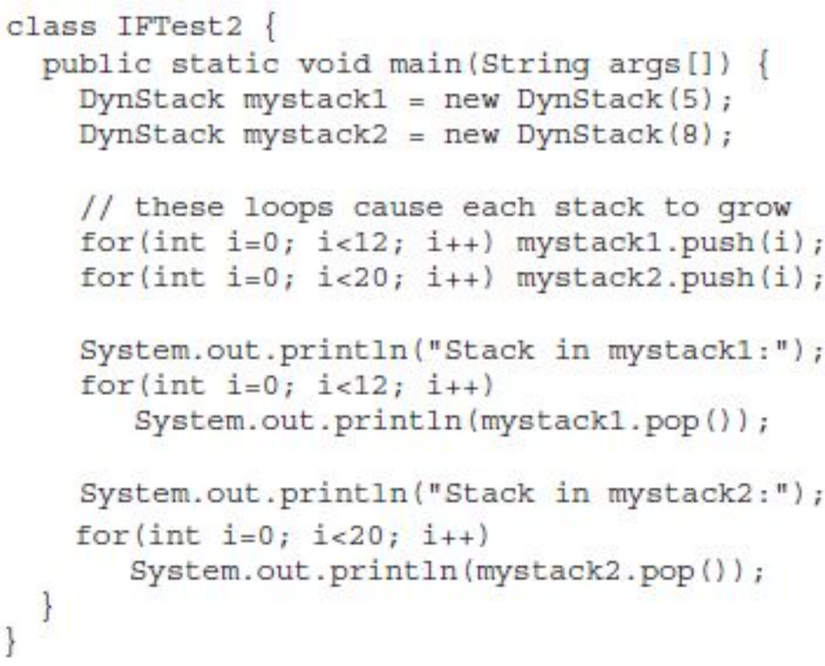
```
class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```

Interfaces

Applying Interfaces: ...



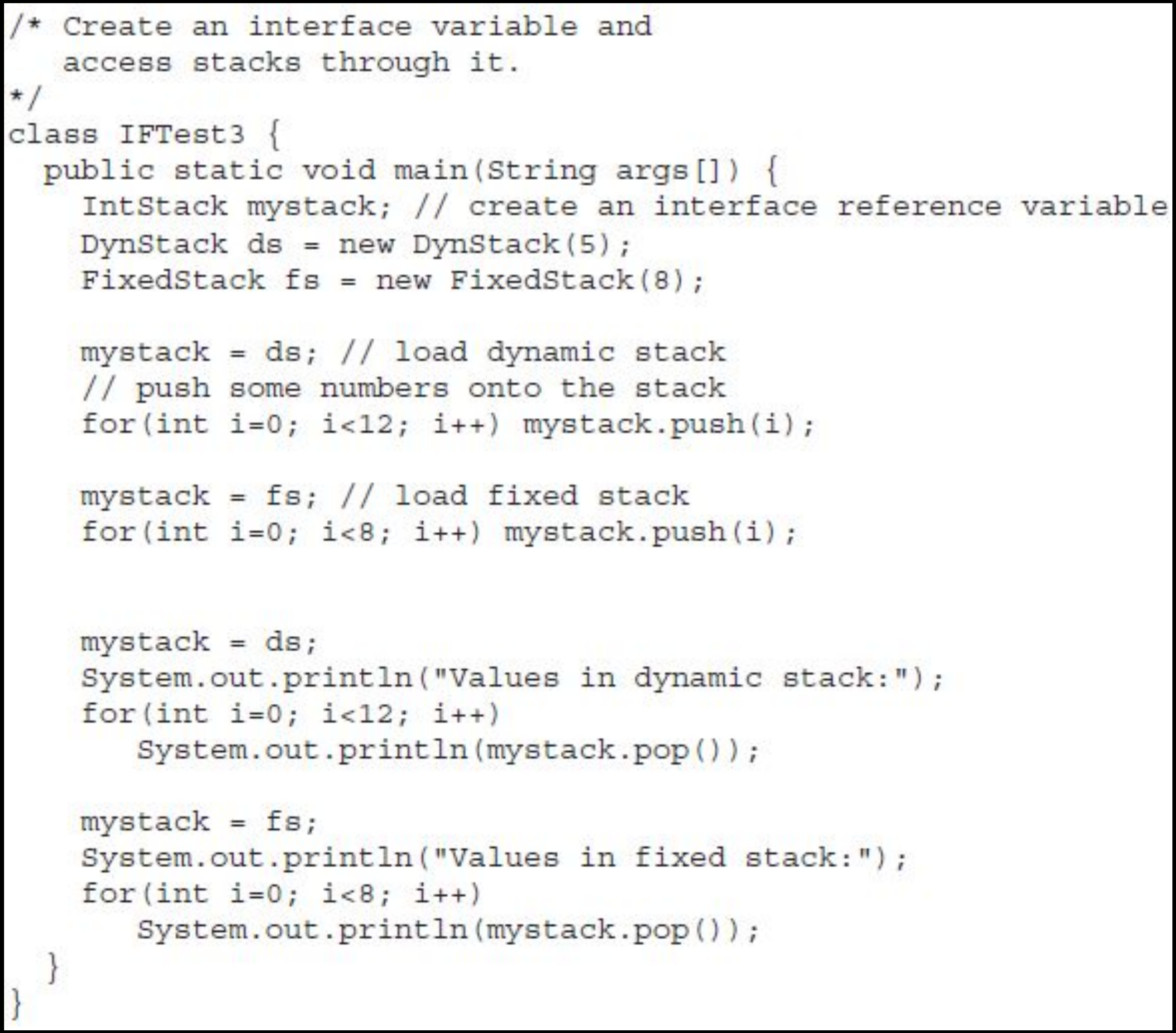
```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```



```
class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // these loops cause each stack to grow
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Stack in mystack1:");
    for(int i=0; i<12; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<20; i++)
       System.out.println(mystack2.pop());
  }
}
```

Interfaces

Applying Interfaces: ...



```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);


    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
       System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
       System.out.println(mystack.pop());
  }
}
```

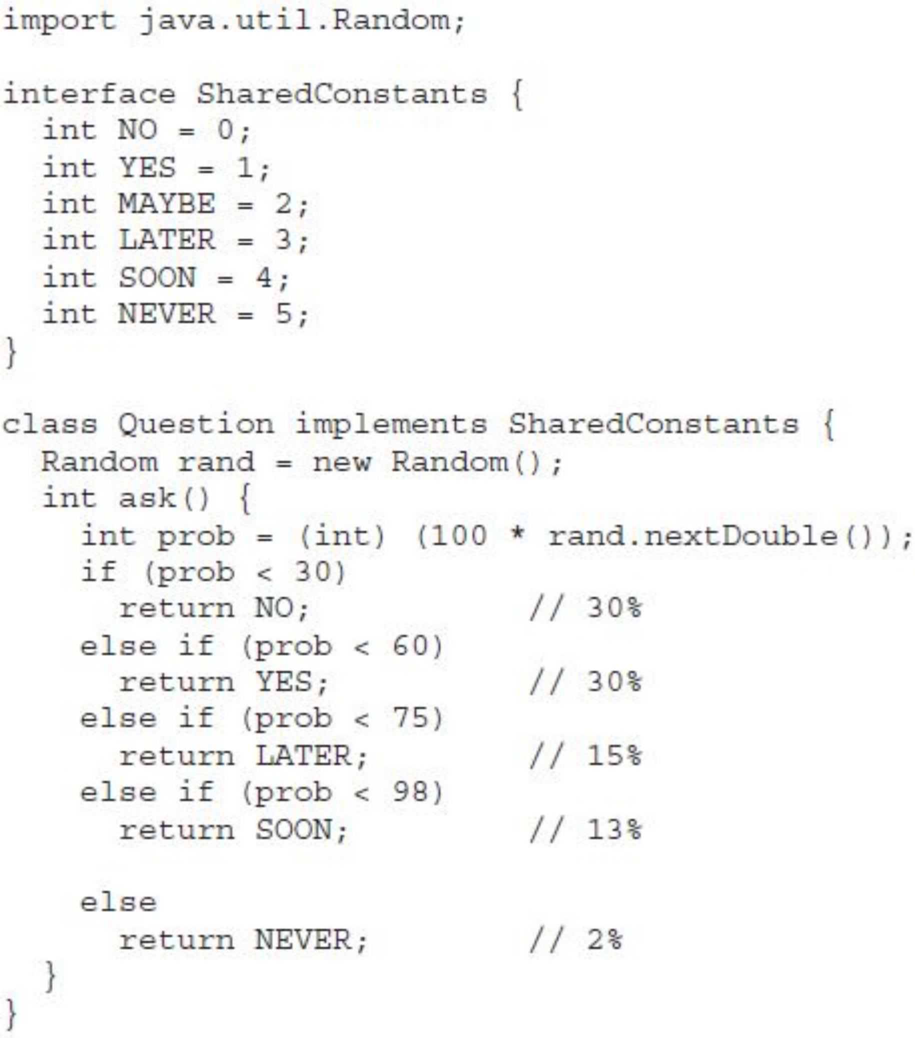
Interfaces

Variables in Interfaces:

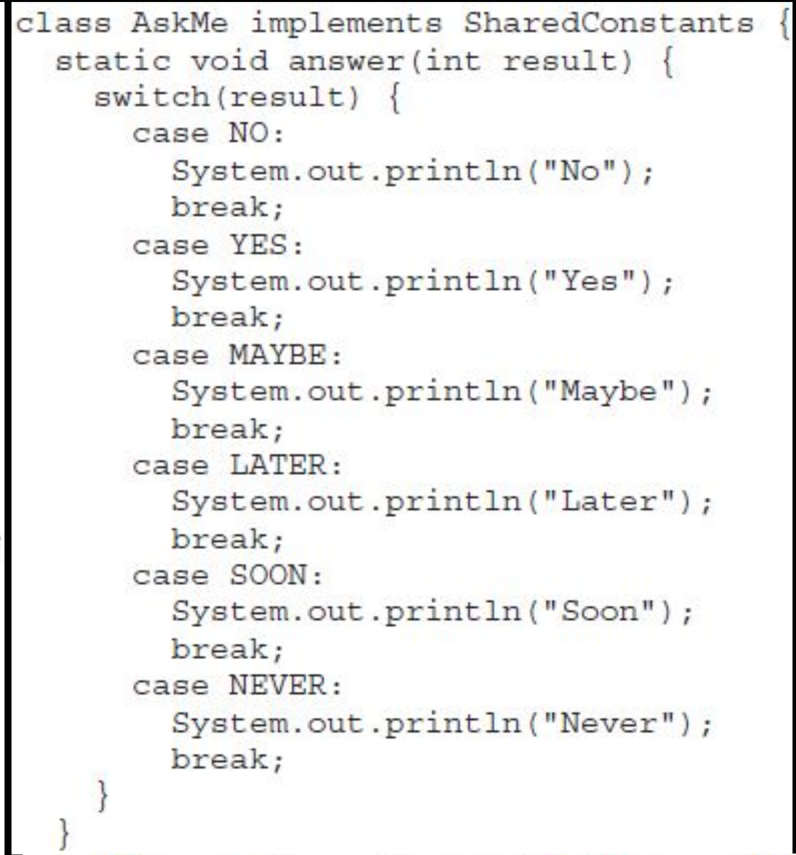
- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants.
- It is as if that class were importing the constant fields into the class name space as **final** variables.

Interfaces

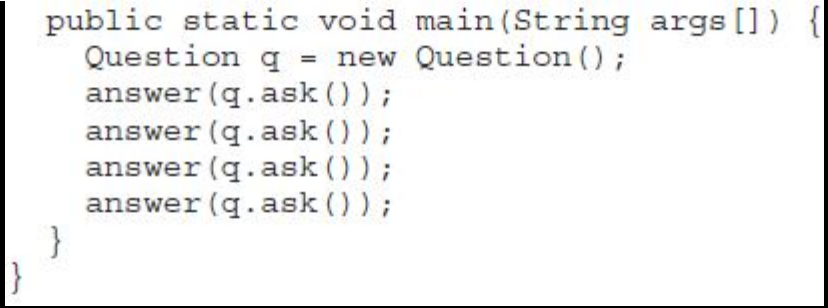
Variables in Interfaces: ...



```
import java.util.Random;

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

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
      return NO;              // 30%
    else if (prob < 60)
      return YES;             // 30%
    else if (prob < 75)
      return LATER;           // 15%
    else if (prob < 98)
      return SOON;            // 13%

    else
      return NEVER;           // 2%
  }
}
```



```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }
```



```
  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

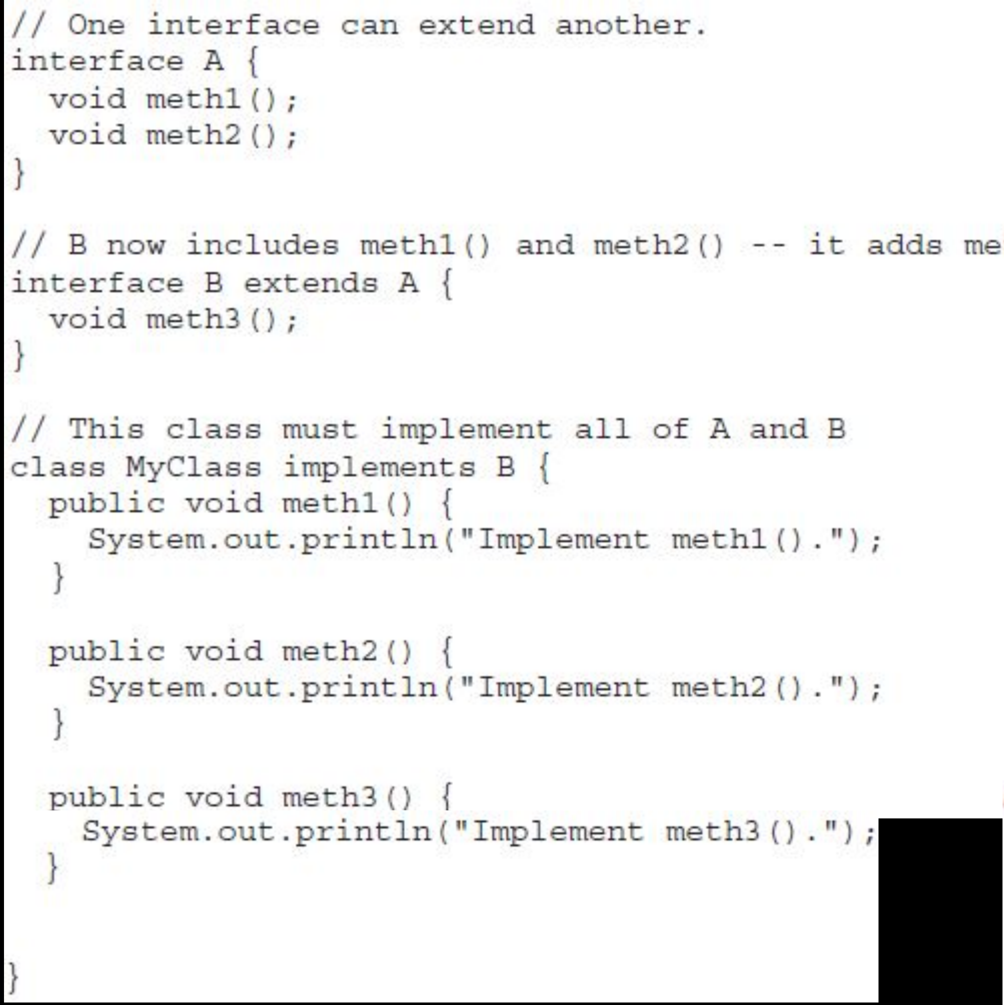
Interfaces

Interfaces Can Be Extended:

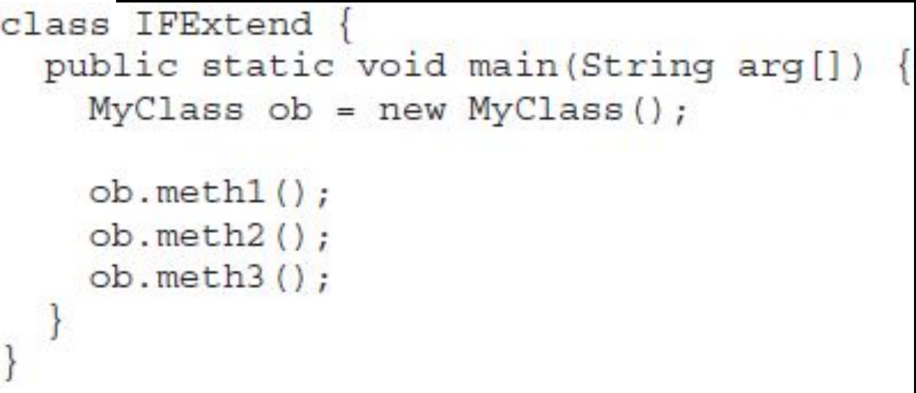
- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Interfaces

Interfaces Can Be Extended:



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

// B now includes meth1() and meth2() -- it adds meth3().
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().");
  }


}
```



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

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

Interfaces

Additional Resources:

Interfaces-1

Interfaces-2

Interfaces-3

Interfaces and Packages

Additional Resources