Programming in Java

Packages and Interfaces

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

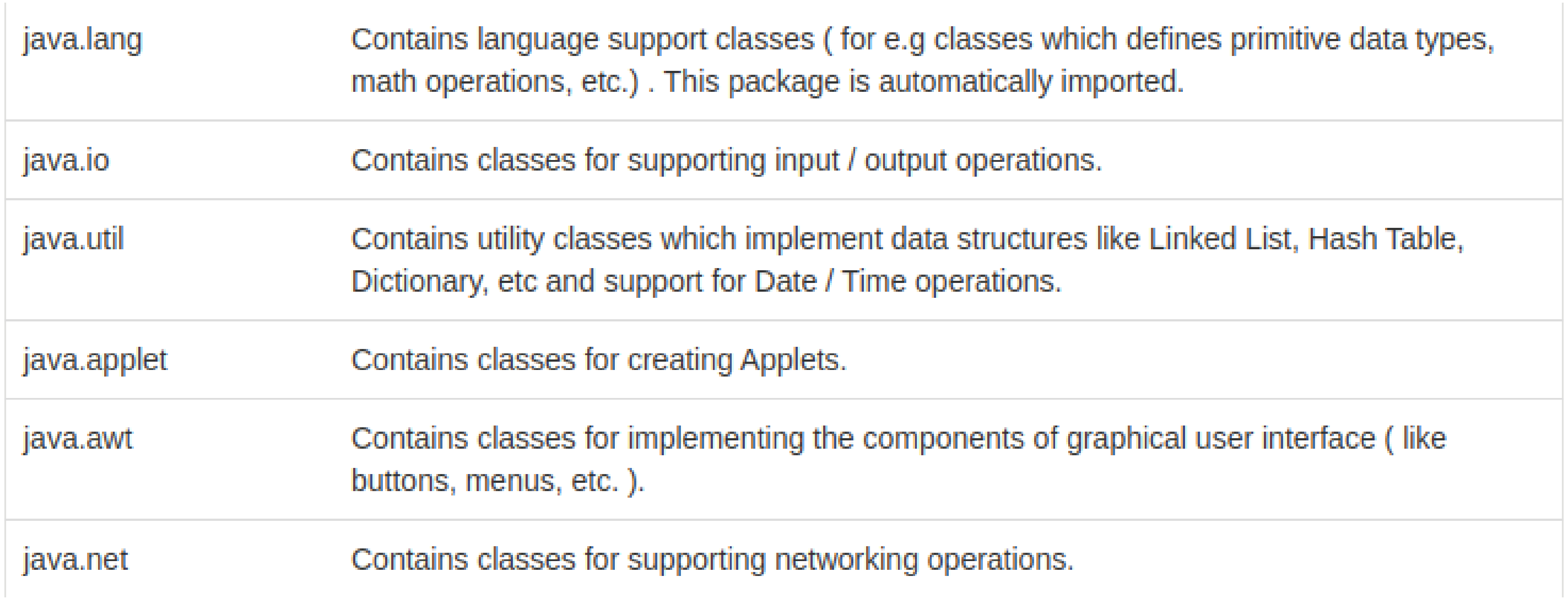
Packages

- A package (act as *Container*) is a collection of related java entities (*such as classes, interfaces, exceptions, errors and enums*), a great way to achieve reusability, can be considered as means to achieve data encapsulation.
- Packages in Java provides a mechanism for partitioning the class name space into more manageable chunks.
- A Package is both a *naming* and a *visibility control* mechanism.
- The advantages of packages are:
  - *Removes naming collision:* by prefixing the class name with a package name.
  - *Provides access control:* Besides *public* and *private*, Java has two access control modifiers- *protected* and *default* (that are related to package).
  - *Categorize the classes and interfaces so that they can be easily maintained.*
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Packages

- Package in java can be categorized in two form, built-in package and user-defined package.

- Built-in packages: standard packages which are part of JRE or Java API. Some of the commonly used built-in packages are:



java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.) . This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).
java.net	Contains classes for supporting networking operations.

Defining a Package

- To create a package is quite easy: simply include a package statement 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, and suitable for short, sample programs but inadequate for real applications.
- Most of the time, for real applications, you will define a package for your code, using the general form:

```
package pkg;            // for example,  package MyPackage;
```

Defining a Package

- *package MyPackage;*   // creates a package called MyPackage;

1. Java uses file system directories to store packages.

2. 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.

3. 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.

4. 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 is:

    package pkg1[.pkg2[.pkg3]];

Defining a Package

- *package MyPackage;* // creates a package called MyPackage;

    5 A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as:

    package java.awt.image; //needs to be stored in java/awt/image in a UNIX environment.

    6 Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

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( ); }
	}
}
```

1 How to compile Java Package program

- *Syntax:* javac -d directory javafilename
- *Example:* javac -d . AccountBalance.java

  //creates package MyPack in current directory (.) and saves the generated .class files(Balance.class and AccountBalance.class) in it. *This step can be performed manually as well.*

2 How to run Java Package program.

- java MyPack.AccountBalance

OUTPUT

```
K. J. Fielding: $123.23
Will Tell: $157.02
--> Tom Jackson: $-12.33
```

** 3 ways are there to locate/run Java Packages program (other two are discussed in next slide)*

Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?

  1. 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.
     - Already Discussed in previous slide

  2. You can specify a directory path or paths by setting the **CLASSPATH** environmental variable. (in Unix/Linux systems)
     - export CLASSPATH=.:/home/UserName/Desktop/MyJavaPrograms;
       // Assuming your packages are saved under Desktop/MyJavaPrograms;

  3. You can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
     - java -classpath /home/UserName/Desktop/MyJavaPrograms/ MyPack.AccountBalance

       // Assuming your packages are saved under Desktop/MyJavaPrograms;
       // Save all .class files of your program(AccountBalance.java) in MyPack.

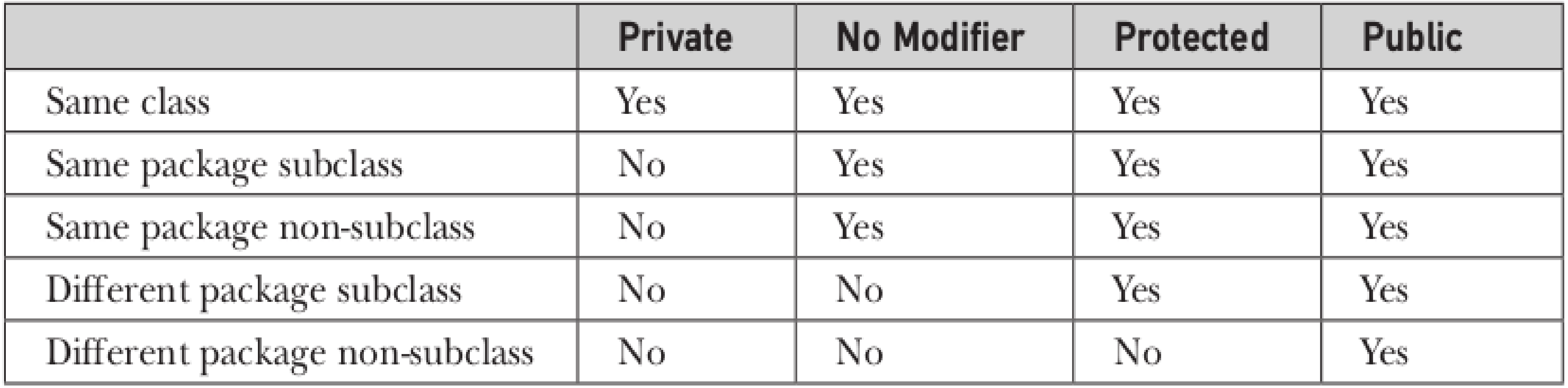
Access Protection

- 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:
  1. *Subclasses in the same package*
  2. *Non-subclasses in the same package*
  3. *Subclasses in different package*
  4. *Classes that are neither in the same package nor subclasses*

Access Protection

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

- The following 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

[Source: (1)]

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 (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.

Access Protection

- A non-nested class has only two possible access levels
  - default and public (others are abstract and final)
- 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

An Access Example

// Shows all combinations of the access control modifiers.

// This example has **two packages** and **five classes.**

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);

		// private member in Protection class
		// System.out.println("n_pri = "+ n_pri);

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

An Access Example

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);
        }
}
```

This is test file for package P1, **DemoP1.java:**

```
// Demo package p1.

package p1;

// Instantiate the various classes in p1.
public class DemoP1 {
        public static void main(String args[ ]) {

                Protection ob1 = new Protection( );

                Derived ob2 = new Derived( );

                SamePackage ob3 = new SamePackage( );
        }
}
```

An Access Example

- How to compile?

  1 Compile all classes one by one in sequence:

  ```
  $ javac -d . Protection.java
  $ javac -d . Derived.java
  $ javac -d . SamePackage.java
  $ javac -d . DemoP1.java
  ```

  OR

  2 Compile all classes all together but in sequence

  ```
  $ javac -d . Protection.java Derived.java SamePackage.java DemoP1.java
  ```

- How to run?

  R

  ```
  $ java p1.DemoP1
  ```

OUTPUT

```
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived constructor
n = 1
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
same package constructor
n = 1
n_pro = 3
n_pub = 4
```

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);
    }
}
```

An Access Example

This is test file for package P2, **DemoP2.java:**

```
// Demo package p2.

package p2;

// Instantiate the various classes in p2.
public class DemoP2 {
        public static void main(String args[ ]) {

                Protection2 ob1 = new Protection2( );

                OtherPackage ob2 = new OtherPackage( );
        }
}
```

OUTPUT

```
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived other package constructor
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
other package constructor
n_pub = 4
```

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. *(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.)*
- The import statement saves a lot of typing. *(If you are going to refer to a few dozen classes in your application)*
- In a Java source file, import statements occur immediately following the packagestatement (if it exists) and before any class definitions.
- The general form of the *import statement:*

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

- For example:

```
import java.util.Date;      //Explicit Date class
import java.io.*;           //Entire io package
```

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

2 There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

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** (*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.;*

- 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:

```
import java.util.*;

class MyDate extends Date {
}
// without the import statement looks like this:
class MyDate extends java.util.Date {   // fully-qualified name
}
```

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.  */

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);
        }
}
```

```
/* Here TestBalance imports MyPack and is then able to
make use of the Balance class:  */

import MyPack.Balance;        //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( );
        }
}
```

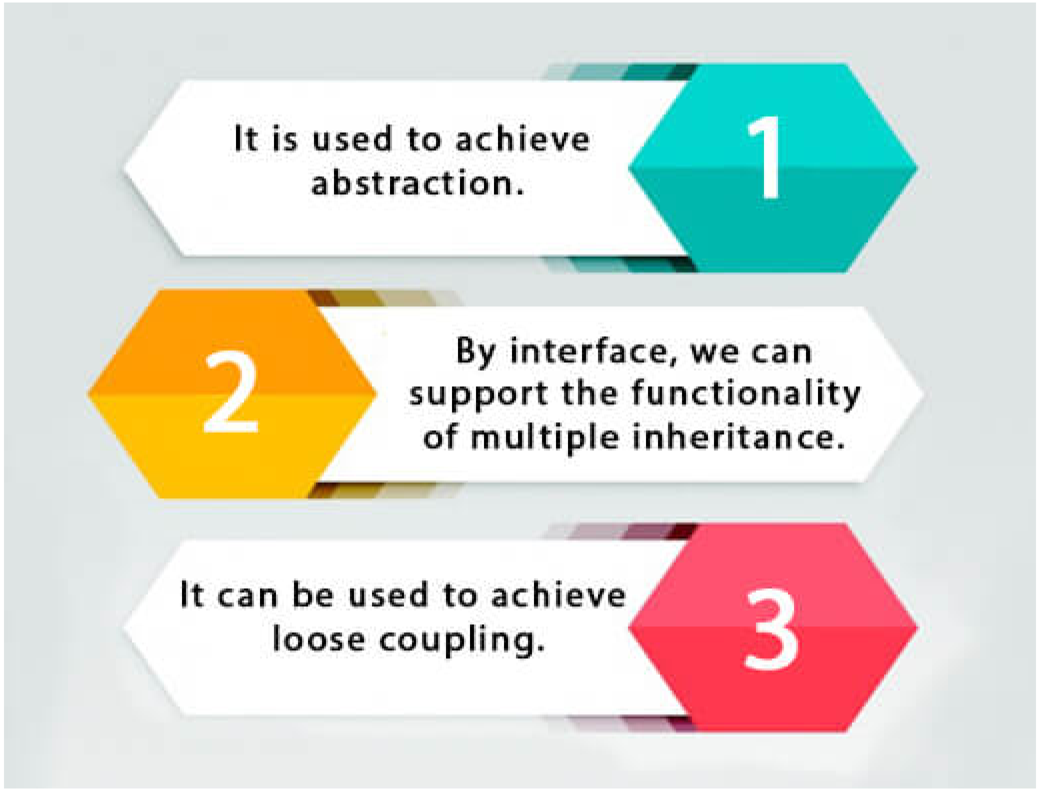
A Remove the public specifier from the Balance class and then try compiling TestBalance.

Interface

- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body(*abstract methods*).
  - *All methods declared in an interface are implicitly public and abstract.*
  - *All variables declared in an interface are implicitly public, static and final.*
- The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- It cannot be instantiated just like the abstract class.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.(*A class can only extend from a single class, but a class can implement multiple interfaces*)
- Interfaces are designed to support dynamic method resolution at run time.

Interface

- Why use interfaces?


1
It is used to achieve abstraction.
2
By interface, we can support the functionality of multiple inheritance.
3
It can be used to achieve loose coupling.

[Source: (3)]

1. *Loose coupling means reducing the dependencies of a class that uses the different classes directly.*
2. *Tight coupling means classes and objects are dependent on one another.*

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;

}
```

1. An interface is declared by using the interface keyword.
2. The access can be either default or public.
3. It provides total abstraction; means all the methods in an interface are declared with the empty body (*Abstract methods*), and all the fields are public, static and final by default.
4. A class that implements an interface must implement all the methods declared in the interface.

Defining an Interface

- Here is an example of an interface definition. It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.

```
interface Callback {

        void callback(int param);

}
```

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
}
```

- 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.

Implementing Interfaces

- 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.
- A sample class implementing the Callback interface shown earlier:

```
class Client implements Callback {

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

- Notice: that callback( ) is declared using the public access modifier.
- REMEMBER: When you implement an interface method, it must be declared as public.

Implementing Interfaces

```
// Finally to test interface

public class InterfaceTest {

    public static void main(String args[ ]) {
        // Can't instantiate an interface directly
        // Callback c1 = new Callback( );
        // c1.callback(21);

        Client c2 = new Client( );

        c2.callback(42);

    }
}
```

What will be the output ?

Implementing Interfaces

- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
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.");
        }
}
```

Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the "callee." (*This process is similar to using a superclass reference to access a subclass object*)

Accessing Implementations Through Interface References

- The following example calls the callback( ) method via an interface reference variable:

```
/* How an interface reference variable
can access an implementation object */

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

                Callback c = new Client( );

                c.callback(42);
        }
}
```

OUTPUT
```
callback called with 42
```

1. Variable **c** is declared to be of the interface type Callback, yet it was assigned an instance of Client.

2. Although **c** can be used to access the callback( ) method, it cannot access any other members of the Client class.

3. An interface reference variable has knowledge only of the methods declared by its interface declaration.

4. Thus, **c** could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

Accessing Implementations Through Interface References

- The following example demonstrate the polymorphic power of interface reference.

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

OUTPUT

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

*The version of callback( ) that is called is determined by the type of object that **c** refers to at run time.*

Partial Implementations

- If a class includes an interface but does not fully implement the methods required 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);
        }

        // no implementation for callback( )
}
```

- 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.

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.

Nested Interfaces

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

1. Class A defines a member interface called NestedIF and that it is declared public.

2. Class B implements the nested interface by specifying:

   implements A.NestedIF

3. Inside the main( ) method, an A.NestedIF reference called nif is created, and it is assigned a reference to a B object.

4. Because B implements A.NestedIF, this is legal.

Applying Interfaces

```
/* Multiple implementations of an interface through an interface reference variable */

interface MyInterface{

        void print(String msg);
}

class MyClass1 implements MyInterface{
        public void print(String msg){
                System.out.println(msg +  " : " +msg.length( ));
        }
}

class MyClass2 implements MyInterface{
        public void print(String msg){
                System.out.println(msg.length( ) + " : " +msg);
        }
}
```

```
Public class InterfaceApplyTest {

        public static void main(String args[ ]) {

                MyClass1 mc1 = new MyClass1( );
                MyClass2 mc2 = new MyClass2( );

                MyInterface mi;  /*create an interface
                                   reference variable */
                mi  = mc1;
                mi.print("Hello World"); // MyClass1 print( )

                mi = mc2;
                mi.print("Hello World"); // MyClass2 print( )
        }
}
```

Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

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.

- *This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.*

- If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables.

Variables in Interfaces

```
// Example to implement an automated "decision maker"
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;
```

Variables in Interfaces

```
        }
    }

    public static void main(String args[ ]) {

        Question q = new Question( );

        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

OUTPUT

```
    Later
    Soon
    No
    Yes
```

Note that the results are different each time it is run.

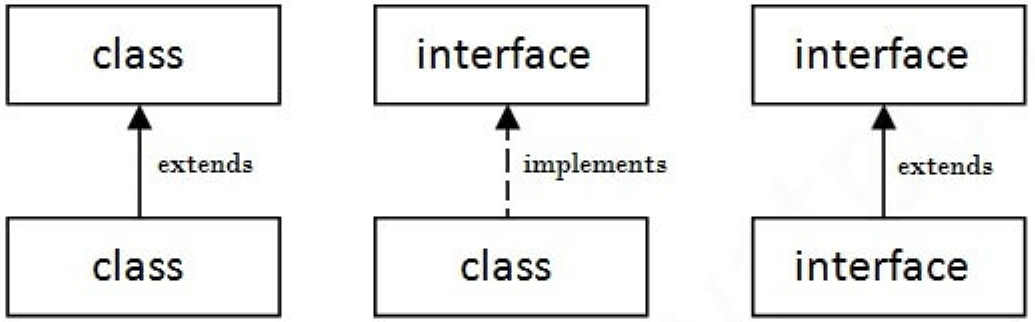
1. This program makes use of one of Java's standard classes: Random.
2. Random contains several methods that allow you to obtain random numbers in the form required by your program.
3. the method nextDouble( ) returns random numbers in the range 0.0 to 1.0.
4. Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined.

Extending Interfaces

- 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 required by the interface inheritance chain.

- A class extends another class, an interface extends another interface, but a class implements an interface.



[Source: (3)]

Extending Interfaces

```
// 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( ).");
        }
}
```

```
class IFExtend {
        public static void main(String arg[ ]) {

                MyClass ob = new MyClass( );

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

Q What will happen if you remove the implementation for meth1( ) in MyClass ?

Extending Interfaces

```
// 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( ).");
        }
}
```

```
class IFExtend {
        public static void main(String arg[ ]) {

                MyClass ob = new MyClass( );

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

Q What will happen if you remove the implementation for meth1( ) in MyClass ?

A This will cause a compile-time error. *Any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.*

Default Interface Methods

- Prior to JDK 8, an interface could not define any implementation whatsoever.
- All the previous versions of Java, the methods specified by an interface were abstract, containing no body.
- The release of JDK 8 has changed this by adding a new capability to interface called the default method.
  - *A default method lets you define a default implementation for an interface method.*
  - *Its primary motivation was to provide a means by which interfaces could be expanded without breaking existing code.*
  - *An interface still cannot have instance variables. The defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class.*
  - *Interfaces that you create will still be used primarily to specify what and not how. However, the inclusion of the default method gives you added flexibility.*

Default Interface Methods

```
//Default interface Method Demo
public interface MyIF {

        int getNumber( );

        // This is a default method. Notice that it provides
        // a default implementation.
        default String getString( ) {
                return "Default String";
        }
}

// Implement MyIF.
class MyIFImp implements MyIF {
        public int getNumber( ) {
                return 100;
        }

        // getString() can be allowed to default.
}
```

```
// Use the default method.
class DefaultMethodDemo {
        public static void main(String args[ ]) {

                MyIFImp obj = new MyIFImp( );

                // Can call getNumber( ), because it is explicitly
                // implemented by MyIFImp:
                System.out.println(obj.getNumber( ));

                // Can also call getString( ), because of default
                // implementation:
                System.out.println(obj.getString( ));
        }
}
```

OUTPUT

```
        100
        Default String
```

Default Interface Methods

- It is both possible and common for an implementing class to define its own implementation of a default method.

```
class MyIFImp2 implements MyIF {

// Here, implementations for both getNumber( ) and getString( ) are provided.

        public int getNumber( ) {
                return 100;
        }
        public String getString( ) {
                return "This is a different string.";
        }
}
```

OUTPUT

```
        100
        This is a different string.
```

Multiple Inheritance Issues

- Java does not support the multiple inheritance of classes, because of ambiguity.
- Default methods do offer a bit of what one would normally associate with the concept of multiple inheritance.
- For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both.
- Thus, to a limited extent, default methods do support multiple inheritance of behavior. But in such a situation, it is possible that a name conflict will occur.
- Observe the code fragments shown in the next slide to understand the scenarios when a name conflict situation may occur.

Multiple Inheritance Issues

```
//Both interfaces define default methods

interface Alpha {
        default void reset( ) {
                System.out.println(“Alpha’s reset”);
        }
}

interface Beta{
        default void reset( ) {
                System.out.println(“Beta’s reset”);
        }
}

class MyClass implements Alpha, Beta {
        public void reset( ) {
                System.out.println(“MyClass’ reset”);
        }
}
```

1. Both Alpha and Beta provide a method called reset() for which both declare a default implementation.

2. Is the version by Alpha or the version by Beta used by MyClass?

Multiple Inheritance Issues

```
//One interfaces extends another, both define default methods.

interface Alpha {
        default void reset( ) {
                System.out.println("Alpha's reset");
        }
}

interface Beta extends Alpha {
        default void reset( ) {

                System.out.println("Beta's reset");
                // Alpha.super.reset( );


        }
}

class MyClass implements Beta{
        public void reset( ) {
                System.out.println("MyClass' reset");
        }
}
```

1 Which version of the default method is used?

2 what if MyClass provides its own implementation of the method?

Q if Beta wants to refer to Alpha's default reset( )

Alpha.super.reset( );

Multiple Inheritance Issues

- To handle previous two cases and other similar types of situations, Java defines a set of rules that resolves such conflicts.

  1. In all cases, a class implementation takes priority over an interface default implementation.
     Ex: if MyClass provides an override of the reset( ) default method, MyClass' version is used.
     Ex: if MyClass implements both Alpha and Beta, both defaults are overridden by MyClass' implementation.

  2. If a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.
     Ex: if MyClass implements both Alpha and Beta, but does not override reset( ), then an error will occur.

  3. If one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence.
     Ex: If Beta extends Alpha, then Beta's version of reset( ) will be used.

  4. It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of super. Its general form is shown here:

```
InterfaceName.super.methodName( )
//Alpha.super.reset( );
```

static Methods in an Interface

- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- Here is the general form:

  InterfaceName.staticMethodName

```
// An example of a static method in an interface
public interface MyIF {

        int getNumber( );

        default String getString( ) {
                return "Default String";
        }

        // This is a static interface method.
        static int getDefaultNumber( ) {
                return 0;
        }
}
```

1. The getDefaultNumber( ) method can be called, as shown here:

   int defNum = MyIF.getDefaultNumber( );

2. No implementation or instance of MyIF is required to call getDefaultNumber( ) because it is static.

3. static interface methods are not inherited by either an implementing class or a subinterface.

References

R **Reference for this topic**

1. **Book-** Java: The Complete Reference, Tenth Edition: Herbert Schildt

2. **Web-** https://www.tutorialspoint.com/java/index.htm

3. **Web-** https://www.javatpoint.com/inheritance-in-java

4. **Web-** https://docs.oracle.com/javase/tutorial/java/IandI/index.html