Unit 2

Packages and Interfaces

Topics Covered

- Interfaces
- Defining an Interface
- Default Interface Methods
- Use static Methods in an Interface

Interfaces

- Using ***interface***, we 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 we 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.
- The keyword ***interface***, can fully abstract a class’ interface from its implementation.
- The interface in Java is a mechanism to achieve ***abstraction***

- 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 to fully utilize the ***"one interface, multiple methods"*** aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- They disconnect the definition of a method or set of methods from the inheritance hierarchy.
- Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

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;
}
```

- When no access modifier 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.
- ***name*** is the name of the interface and can be any valid identifier.
- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods.
- Each class that includes such an interface must implement all of the methods.
- As the general form shows, 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***.

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.
- 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.

Here is a small example class that implements 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);
}
}
```

Note : When we 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.

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.");
} }
```

Java Example program using 'interface'

```
interface printable
{
void print();
}
class A6 implements printable
{
public void print()
{
System.out.println("Hello");
}

public static void main(String args[])
{
A6 obj = new A6();
obj.print();
 }
}
```

Accessing Implementations Through Interface References

- We 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 we 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.”

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

```
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
```

The variable ***c*** is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although ***c*** can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable has knowledge only of the methods declared by its interface declaration. Thus, ***c*** could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

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***.

For example:

```
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.

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.

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

implements A.NestedI

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.

Variables in Interfaces

- We 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 we include that interface in a class (that is, when we “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.

Example Program 1

```
interface SampleInterface
{
	int LIMIT = 100;
}

class InterfaceVariablesExample implements SampleInterface
{
	public static void main(String[] args)
	{
		System.out.println("Maximum Limit = " +LIMIT);
	}
}
```

Output of the program

Maximum Limit= 100

Example Program 2

The following example uses this technique 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;
    }
  }

  public static void main(String args[]) {
    Question q = new Question();

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

This program makes use of one of Java's standard classes: ***Random***. This class provides pseudorandom numbers. It contains several methods to obtain random numbers.In this example, the method ***nextDouble( )*** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, ***Question*** and ***AskMe***, both implement the ***SharedConstants*** interface where NO, YES, MAYBE, SOON, LATER and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly.

Here is the output of a sample run of this program. Note that the results are different each time it is run.

Later
Soon
No
Yes

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 required by the interface inheritance chain.

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

Default Interface Methods

- The release of JDK 8 has changed the traditional form of interface by adding a new capability to interface called the ***default method***.
- A default method define a default implementation for an interface method.
- In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract.
- ***A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code***. There must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause pre existing code to break.

- ***Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.*** For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called remove( ), and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences,then remove( ) is essentially optional because it won't be used by non-modifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of remove( ), even though it was not needed. Today, a default implementation for remove( ) can be specified in the interface that does nothing. Providing this default prevents a class used for non-modifiable sequences from having to define its own, placeholder version of remove( ). Thus, by providing a default, the interface makes the implementation of remove( ) by a class optional.

Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a class. The primary difference is that the declaration is preceded by the keyword ***default***.

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

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

MyIF declares two methods. The first, ***getNumber( )***, is a standard interface method declaration. It defines no implementation whatsoever. The second method is ***getString( )*** and it does include a default implementation. In this case, it simply returns the string "Default String". Its declaration is preceded by the default modifier. To define a default method, precede its declaration with ***default***.

Because ***getString( )*** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the ***MyIFImp*** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

The following code creates an instance of MyIFImp and uses it to call both ***getNumber( )*** and ***getString( )***.

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

The output is shown here:

100
Default String

The default implementation of ***getString( )*** was automatically used. It was not necessary for ***MyIFImp*** to define it. Thus, for ***getString( )***, implementation by a class is optional.

It is both possible and common for an implementing class to define its own implementation of a default method. For example, ***MyIFImp2*** overrides ***getString( )***

```
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.";
  }
}
```

Now, when **getString( )** is called, a different string is returned.

Multiple Inheritance Issues

Consider 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. In such a situation, it is possible that a name conflict will occur.

- For example, assume that two interfaces called ***Alpha*** and ***Beta*** are implemented by a class called ***MyClass***. What happens if both ***Alpha*** and ***Beta*** provide a method called ***reset( )*** for which both declare a default implementation? Is the version by ***Alpha*** or the version by ***Beta*** used by ***MyClass***?
- Or, consider a situation in which ***Beta*** extends ***Alpha.*** Which version of the default method is used?
- Or, what if ***MyClass*** provides its own implementation of the method?

To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

- First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if ***MyClass*** provides an override of the ***reset( )*** default method, ***MyClass*** version is used. This is the case even if ***MyClass*** implements both ***Alpha*** and ***Beta***. In this case, both defaults are overridden by ***MyClass***' implementation.
- Second, in cases in which 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. Continuing with the example, if ***MyClass*** implements both ***Alpha*** and ***Beta***, but does not override ***reset( ),*** then an error will occur.
- In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if ***Beta*** extends ***Alpha***, then Beta's version of ***reset( )*** will be used.

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( )

For example, if Beta wants to refer to Alpha's default for reset( ), it can use this statement:

Alpha.super.reset();

Use static Methods in an Interface

- JDK 8 added new capability to interface: the ability to define one or more static methods.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- Thus, no implementation of the interface is necessary and no instance of the interface is required, in order to call a static method.
- Instead, a static method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a static method in a class is called.

The following shows an example of a static method in an interface by adding one to ***MyIF***. The static method is ***getDefaultNumber( )***. It returns zero.

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
   // This is a static interface method
   static int getDefaultNumber() {
     return 0;
   }
}
```

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

int defNum = MyIF.getDefaultNumber();

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

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