Module – 3

Interfaces

Interfaces

Using ***interface,*** you can fully abstract a class' interface from its implementation. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, 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. To implement an interface, a class must provide the complete set of methods required by the interface.

Defining 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;

}

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

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

}

Here is a small example class that implements the Callback interface,

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

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

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

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 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 Notice that 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.

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.

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, as previously described. 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. Here is an example that demonstrates a nested interface:

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

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.

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. Following is an example:

```
// 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[] args) {
    MyClass ob = new MyClass();

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

Default Interface Methods

A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an extension method. The default method provides a means by which interfaces could be

expanded without breaking existing code. 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. For example, consider this simple interface:

```
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";
  }
}
```

Another example,

```
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item

  // Because clear( ) has a default, it need not be
  // implemented by a preexisting class that uses IntStack.
  default void clear() {
    System.out.println("clear() not implemented.");
  }
}
```

The default method gives

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

Use static Methods in an Interface

Another capability added to interface by JDK 8 is 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, shown in the previous section. 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;
  }
}
```

Private Interface Methods

An interface can include a private method. A private interface method can be called only by a default method or another private method defined by the same interface. Because a private interface method is specified private, it cannot be used by code outside the interface in which it is defined. This restriction includes subinterfaces because a private interface method is not inherited by a subinterface. The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication.

For example, here is version of the IntStack interface that has two default methods called popNElements( ) and skipAndPopNElements( ). The first returns an array that contains the top N elements on the stack. The second skips a specified number of elements and then returns an array that contains the next N elements. Both use a private method called getElements( ) to obtain an array of the specified number of elements from the stack.

```
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item

// A default method that returns an array that contains
// the top n elements on the stack.
default int[] popNElements(int n) {
  // Return the requested elements.
  return getElements(n);
}

// A default method that returns an array that contains
// the next n elements on the stack after skipping elements.
default int[] skipAndPopNElements(int skip, int n) {

  // Skip the specified number of elements.
  getElements(skip);

  // Return the requested elements.
  return getElements(n);
}

// A private method that returns an array containing
// the top n elements on the stack
private int[] getElements(int n) {
  int[] elements = new int[n];

  for(int i=0; i < n; i++) elements[i] = pop();
  return elements;
}
}
```