6.2: Interfaces

IT1406 - Introduction to Programming

Level I - Semester 1

6.2. Interfaces

6.2. Interfaces

- Through the use of the interface keyword, Java allows you to fully abstract an interface from its implementation.
- Using interface, you can specify a set of methods that can be implemented by one or more classes.
- In its traditional form, 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).

6.2. Interfaces

- 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. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.
- This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy.

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;
}
```

Defining an interface (contd.)

- 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.
- Default methods constitute what is, in essence, a special-use feature, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no default methods exist. For this reason, we will begin by discussing the interface in its traditional form.

Defining an interface (contd.)

- The default method is described at the end of this chapter. 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**.
- 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.

Implementing interfaces (contd.)

- 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);
    }
}
```

Implementing interfaces (contd.)

- **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. 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 (contd.)

- 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
```

Accessing Implementations Through Interface References (contd.)

- 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, 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:

Nested interfaces (contd.)

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

Nested interfaces (contd.)

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

  implements A.NestedIF

- 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.

Applying interfaces

- There are many ways to implement a stack. For example, the stack can be of a fixed size or it can be “growable.”
- The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, 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. Let’s look at two examples.

Applying interfaces (contd.)

- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

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

- The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

Applying interfaces (contd.)

```
// 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.");
```

Applying interfaces (contd.)

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

Applying interfaces (contd.)

- Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// 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
```

Applying interfaces (contd.)

```
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
```

Applying interfaces (contd.)

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

Applying interfaces (contd.)

- The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.
- In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Applying interfaces (contd.)

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

Variables in interfaces

- 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.
- 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. The next example uses this technique to implement an automated "decision maker":

Variables in interfaces (contd.)

```
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;
```

Variables in interfaces (contd.)

```
        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;
```

Variables in interfaces (contd.)

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

Variables in interfaces (contd.)

- 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.
- 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();
}
```

Interfaces can be extended (contd.)

```
// 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

- Prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all 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.
- 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. During its development, the default method was also referred to as an *extension method,* and you will likely see both terms used

Default interface methods (contd.)

- It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example.
- 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";
        }
}
```

Default interface methods (contd.)

- **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". Pay special attention to the way **getString( )** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. 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:

Default interface methods (contd.)

```
// 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( )**.

Default interface methods (contd.)

```
// 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 :

```
100
Default String
```

Default interface methods (contd.)

- As you can see, 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. (Of course, its implementation by a class will be *required* if the class uses **getString( )** for some purpose beyond that supported by its default.)
- 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

- Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.
- The preceding notwithstanding, 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. As you might guess, in such a situation, it is possible that a name conflict will occur.

Multiple Inheritance Issues

- 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.

Multiple Inheritance Issues

- 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.

Multiple Inheritance Issues (contd.)

- 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();

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.
- 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.

Static methods in an interface (contd.)

- 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;
    }
}
```

Static methods in an interface (contd.)

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

```
int defNum = MyIF.getDefaultNumber();
```

- As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.
- One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.