Java Lecture-10

Interface

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.

- That is, using **interface**, you can specify what a class must do, but not how it does it.

- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.

- Once an interface 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.

- However, each class is free to determine the details of its own implementation.

- By providing the **interface** keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

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.

- The methods that are declared have no bodies. They end with a semicolon after the parameter list.

- Each class that includes such an interface must implement all of the methods.

- Prior to JDK 8, an interface could define only “what,” but not “how.” JDK 8 changes this. Beginning with JDK 8, it is possible to add a *default implementation* to an interface method.

- 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);
}
```

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.

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

```
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.

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

Accessing Implementations Through Interface References

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

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

- 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.

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.

- 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.

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.