Standard Interfaces in Java

Some of Java's Most used interfaces

Iterator	• To run through a collection of objects without knowing how the objects are stored, for example, in array, list, bag, or set.
Cloneable	• To make a copy of an existing object via the `clone()` method on the class `Object`.
Serializable	• Pack a web of objects such that it can be send over a network or stored to disk. A naturally later be restored as a web of objects `Comparable`.
Comparable	• To make a total order of objects, for example, 3, 56, 67, 879, 3422, 34234

Iterator interface

The Iterator interface in the package **java.util** is a basic iterator that works on collections.

```
package java.util.*;
public interface Iterator {
     public abstract boolean hasNext();  // Check, if the list has more
     Object next();    // Return the next element
     void remove(); // optional throws exception
 }
     // use an iterator
      myShapes = getSomeCollectionOfShapes(); // Has set of objects
      Iterator iter = myShapes.iterator();
          while (iter.hasNext()) {
            Shape s = (Shape)iter.next(); // downcast
            s.draw();
      }
```

Iterator interface

The Iterator interface in the package **java.util** is a basic iterator that works on collections.

```
package java.util.*;
public interface Iterator {
    public abstract boolean hasNext();  // Check, if the list has more
    Object next();    // Return the next element
    void remove(); // optional throws exception
}
    // use an iterator
     myShapes = getSomeCollectionOfShapes(); // Has set of objects
     Iterator iter = myShapes.iterator();
         while (iter.hasNext()) {
             Shape s = (Shape)iter.next(); // downcast
             s.draw();
         }
```

Cloneable interface

- A class `X` that implements the `Cloneable` interface tells that the objects of class `X` can be cloned.
- The interface is empty, that is, it has no method.
- Returns an identical copy of an object.
  - A shallow copy, by default.
  - A deep copy is often preferable.
- Prevention of cloning
  - Necessary, if unique attribute, for example, database lock or open file reference.
  - Not sufficient to omit to implement `Cloneable`.
    - Sub classes might implement it.
  - Clone method should throw an exception:
    - `CloneNotSupportedException`

Cloneable Interface: Example

```
public class Car implements Cloneable{
      private String make;
      private String model;
      private double price;
      public Car() {// default constructor
            this("", "", 0.0);
      }
 // give reasonable values to instance variables
      public Car(String make, String model, double price){
            this.make = make;
            this.model = model;
            this.price = price;
      }
      public Object clone(){// the Cloneable interface
            return new Car(this.make, this.model, this.price);
    }
}
```

Cloneable Interface: Example

```
public class Car implements Cloneable{
      private String make;
      private String model;
      private double price;
      public Car() {// default constructor
            this("", "", 0.0);
      }
 // give reasonable values to instance variables
      public Car(String make, String model, double price){
            this.make = make;
            this.model = model;
            this.price = price;
      }
      public Object clone(){// the Cloneable interface
            return new Car(this.make, this.model, this.price);
      }
}
```

Serializable interface

```
public class Car implements Serializable {
        // rest of class unaltered

}
      // write to and read from disk
import java.io.*;
public class SerializeDemo{
        Car myToyota, anotherToyota;
        myToyota = new Car("Toyota", "Carina", 42312);
        ObjectOutputStream out = getOutput();
        out.writeObject(myToyota);
        ObjectInputStream in = getInput();
        anotherToyota = (Car)in.readObject();
}
```

Serializable interface

```
public class Car implements Serializable {
        // rest of class unaltered

}
      // write to and read from disk
import java.io.*;
public class SerializeDemo{
        Car myToyota, anotherToyota;
        myToyota = new Car("Toyota", "Carina", 42312);
        ObjectOutputStream out = getOutput();
        out.writeObject(myToyota);
        ObjectInputStream in = getInput();
        anotherToyota = (Car)in.readObject();
}
```

- ✓ A class X that implements the Serializable interface tells clients that X objects can be stored on a file or other persistent me
- ✓ The interface ty, that is, has no metho

Comparable interface: Example

```
public class IPAddress implements Comparable{
    private int[] n; // here IP stored, e.g., 125.255.231.123
    /** The Comparable interface */
    public int compareTo(Object o){
        IPAddress other = (IPAddress) o; // downcast
        int result = 0;
        for(int i = 0; i < n.length; i++){
            if (this.getNum(i) < other.getNum(i)){
                result = -1;
                break;
            }
            if (this.getNum(i) > other.getNum(i)){
                result = 1;
                break;
            }
        }
        return result;
    }
}
```

Some Salient Points

Defining an interface

- Defining an interface is similar to creating a new class.
- An interface definition has two components: the interface declaration and the interface body.

```
interfaceDeclaration
  {
        interfaceBody
  }
```

The `interfaceDeclaration` declares various attributes about the interface such as its name and whether it extends another interface, etc.

- The `interfaceBody` contains the constant and method declarations within the interface.

Defining an interface

```
public interface StockWatcher
{
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged (String tickerSymbol, double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface.

Defining an interface

```
public interface StockWatcher
{
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged (String tickerSymbol, double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface.

Implementing an 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
 }
```

- 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 that method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.

Implementing an 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
 }
```

- 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 that method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.

Implementing interfaces: An example

Example: A class that implements, say Callback interface:

```
class Client implements Callback {
    // Implement Callback's interface

    public void callback(int p) {
        System.out.println("callback called with " + p);
    }
}
```

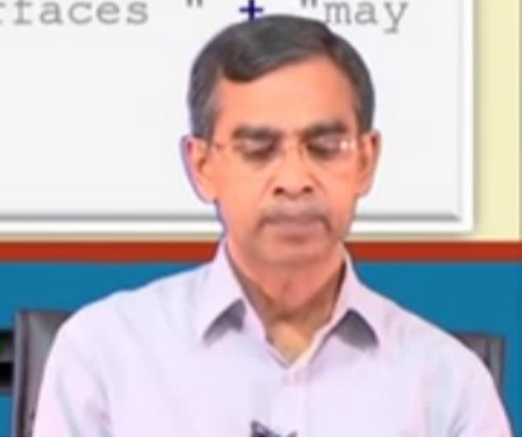
When you implement an interface method, it must be declared as `public`

Implementing interfaces: An example

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

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.");
    }
}
```



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.

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.

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.

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 nested interface.

A nested interface can be declared as public, private, or protected.

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.

Nested interfaces: Example

```
// This class contains a nested interface.
class A {
  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");
 } }
```

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

B implements the nested interface by specifying implements *A.NestedIF*

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*.

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.

```
import java.util.Random;
    interface SharedConstants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
```

This program makes use of one of Java's standard classes: Random, which provides pseudorandom numbers.

Variables in interfaces

```
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%
     }
}
```

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 classes, **Question** implements the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside the class, the code refers to these constants as if each class had defined or inherited them directly.

Variables in interfaces

```
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%
        }
}
```

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 classes, **Question** implements the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside the class, the code refers to these constants as if each class had defined or inherited them directly.

Variables in interfaces

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

Interfaces can be extended

- An interface can inherit another using the keyword **extends**. The syntax is the same as for inheriting classes.

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

When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Interfaces can be extended

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

As an experiment, if you try removing the implementation for **meth1( )** in MyClass, it will cause a compile-time error.

Multiple Inheritance Issue

Multiple inheritance issues

- Java does not support the multiple inheritance of classes. There is 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.

- For example, assume that two interfaces called `Alpha` and `Beta` are implemented by a class called `MyClass`. What happens if both `Alpha` and `Beta` declare 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

Rules.

- First, in all cases, a class implementation takes priority over an interface default implementation.
  - Thus, if `MyClass` provides an override of the `reset( )` method, `MyClass`' version is used.
  - This is the case even if `MyClass` implements, say 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 occurs. Continuing with the example, if `MyClass` implements both `Alpha` and `Beta`, but does not override `reset( )`, then an error will occur.

Multiple inheritance issues

Rules.

- In cases, 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:

Multiple Inheritance Issues

- 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();
```

Multiple Inheritance Issues

- 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();
```

Questions to think...

- How a robust program can be developed in Java ?
- How Java manages different types of errors in programs so that it can avoid abnormal termination of programs?

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

OBJECT ORIENTED PROGRAMMING WITH JAVA

Interface: Demonstration - IX

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

In today's demonstration

1. Creating an interface.
2. Some properties of interface.
3. Interface and single inheritance.
4. Interface and multiple inhritance.
5. Runtime polymorphism with interface
6. Interface versus abstract class