

Interfaces

- In object-oriented programming, it is sometimes helpful to define what a class must do but not how it will do it.
- You have already seen an example of this: the abstract method.
- An abstract method defines the signature for a method but provides no implementation.
- A subclass must provide its own implementation of each abstract method defined by its superclass.

Interfaces

- Thus, an abstract method specifies the *interface* to the method but not the *implementation*.
- While abstract classes and methods are useful, it is possible to take this concept a step further.
- In Java, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces

- An **interface** is syntactically similar to an abstract class, in that you can specify one or more methods that have no body.
- Those methods must be implemented by a class in order for their actions to be defined.
- Thus, an interface specifies what must be done, but not how to do it.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement any number of interfaces.

Interfaces

- By providing the **interface** keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.
- **JDK 8** added a feature to interface that made a significant change to its capabilities.
- However, such methods constitute what are, in essence, **special-use features**, and the original intent behind **interface** still remains.

Interfaces

- For this reason, we will begin by discussing the interface in its traditional form.
- The expanded interface features are described at the end of this chapter.

Interfaces

- Here is a simplified general form of a traditional interface:

```
access interface name {  
    ret-type method-name1(param-list);  
    ret-type method-name2(param-list);  
    type var1 = value;  
    type var2 = value;  
    // ...  
    ret-type method-nameN(param-list);  
    type varN = value;  
}
```

Interfaces

- For a top-level interface, access is either **public** or not used.
- When no access modifier is included, then default access results, and the interface is available only to other members of its package.
- When it is declared as public, the interface can be used by any other code.
- (When an interface is declared public, it must be in a file of the same name.) *name* is the name of the interface and can be any valid identifier.

Interfaces

- In the traditional form of an interface, methods are declared using only their return type and signature.
- They are, essentially, abstract methods.
- Thus, each class that includes such an **interface** must implement all of its methods.
- In an interface, methods are implicitly **public**.

Interfaces

- Variables declared in an **interface** are not instance variables.
- Instead, they are implicitly public, final, and static and must be initialized. Thus, they are essentially constants.

Interfaces

- Here is an example of an **interface** definition.
- It specifies the interface to a class that generates a series of numbers.

```
public interface Series {  
    int getNext(); // return next number in series  
    void reset(); // restart  
    void setStart(int x); // set starting value  
}
```

- This interface is declared **public** so that it can be implemented by code in any package.

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:

Implementing Interfaces

```
class classname extends superclass implements interface {  
    // class-body  
}
```

- To implement more than one interface, the interfaces are separated with a comma. Of course, the **extends** clause is optional.

Implementing Interfaces

- 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 an example that implements the **Series** interface shown earlier.
- It creates a class called **ByTwos**, which generates a series of numbers, each two greater than the previous one.

Implementing Interfaces

```
// Implement Series.  
class ByTwos implements Series {  
    int start;  
    int val;  
  
    ByTwos() {  
        start = 0;  
        val = 0;  
    }  
  
    public int getNext() {  
        val += 2;  
        return val;  
    }  
}
```

↑
Implement the **Series** interface.

Implementing Interfaces

```
public void reset() {  
    val = start;  
}  
  
public void setStart(int x) {  
    start = x;  
    val = x;  
}  
}
```

- Notice that the methods **getNext()**, **reset()**, and **setStart()** are declared using the **public** access specifier.
- This is necessary.

Implementing Interfaces

- Whenever you implement a method defined by an interface, it must be implemented as **public** because all members of an interface are implicitly **public**.

Implementing Interfaces

- Here is a class that demonstrates **ByTwos**:

Implementing Interfaces

```
class SeriesDemo {
    public static void main(String[] args) {
        ByTwos ob = new ByTwos();

        for(int i=0; i < 5; i++)
            System.out.println("Next value is " +
                               ob.getNext());

        System.out.println("\nResetting");
        ob.reset();
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " +
                               ob.getNext());
        System.out.println("\nStarting at 100");
        ob.setStart(100);
        for(int i=0; i < 5; i++)
            System.out.println("Next value is " +
                               ob.getNext());
    }
}
```

Implementing Interfaces

The output from this program is shown here:

```
Next value is 2  
Next value is 4  
Next value is 6  
Next value is 8  
Next value is 10
```

```
Resetting  
Next value is 2  
Next value is 4  
Next value is 6  
Next value is 8  
Next value is 10
```

```
Starting at 100  
Next value is 102  
Next value is 104  
Next value is 106  
Next value is 108  
Next value is 110
```

Implementing Interfaces

- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of **ByTwos** adds the method **getPrevious()**, which returns the previous value:

Implementing Interfaces

```
// Implement Series and add getPrevious().
class ByTwos implements Series {
    int start;
    int val;
    int prev;

    ByTwos() {
        start = 0;
        val = 0;
        prev = -2;
    }

    public int getNext() {
        prev = val;
        val += 2;

        return val;
    }
}
```

Implementing Interfaces

```
public void reset() {  
    val = start;  
    prev = start - 2;  
}  
  
public void setStart(int x) {  
    start = x;  
    val = x;  
    prev = x - 2;  
}  
  
int getPrevious() { ← Add a method not defined by Series.  
    return prev;  
}  
}
```

Implementing Interfaces

- As explained, any number of classes can implement an **interface**. For example, here is a class called **ByThrees** that generates a series that consists of multiples of three:

```
// Implement Series.  
class ByThrees implements Series { ← Implement Series a different way.  
    int start;  
    int val;  
  
    ByThrees() {  
        start = 0;  
        val = 0;  
    }  
  
    public int getNext() {  
        val += 3;  
        return val;  
    }  
}
```

Implementing Interfaces

```
public void reset() {  
    val = start;  
}  
  
public void setStart(int x) {  
    start = x;  
    val = x;  
}  
}
```


Implementing Interfaces

- ***One more point: If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared abstract.
- No objects of such a class can be created, but it can be used as an abstract superclass, allowing subclasses to provide the complete implementation.

Using Interface References

- you can declare a reference variable of an interface type.
- In other words, you can create an interface reference variable.
- Such a variable can refer to any object that implements its interface.
- When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed.

Using Interface References

- This process is similar to using a superclass reference to access a subclass object.
- The following example illustrates this process.
- It uses the same interface reference variable to call methods on objects of both **ByTwos** and **ByThrees**.

Using Interface References

```
// Demonstrate interface references.
```

```
class ByTwos implements Series {  
    int start;  
    int val;  
  
    ByTwos() {  
        start = 0;  
        val = 0;  
    }  
  
    public int getNext() {  
        val += 2;  
        return val;  
    }  
  
    public void reset() {  
        val = start;  
    }  
}
```

Using Interface References

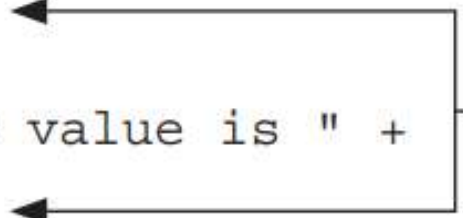
```
public void setStart(int x) {  
    start = x;  
    val = x;  
}  
}  
  
class ByThrees implements Series {  
    int start;  
    int val;  
  
    ByThrees() {  
        start = 0;  
        val = 0;  
    }  
}
```

Using Interface References

```
public int getNext() {  
    val += 3;  
    return val;  
}  
  
public void reset() {  
    val = start;  
}  
  
public void setStart(int x) {  
    start = x;  
    val = x;  
}  
}
```

Using Interface References

```
class SeriesDemo2 {  
    public static void main(String[] args) {  
        ByTwos twoOb = new ByTwos();  
        ByThrees threeOb = new ByThrees();  
        Series ob;  
  
        for(int i=0; i < 5; i++) {  
            ob = twoOb;  
            System.out.println("Next ByTwos value is " +  
                               ob.getNext());  
  
            ob = threeOb;  
            System.out.println("Next ByThrees value is " +  
                               ob.getNext());  
        }  
    }  
}
```



Access an object via
an interface reference.

Variables in Interfaces

- As mentioned, variables can be declared in an interface, but they are implicitly **public**, **static**, and **final**.
- Large programs typically make use of several constant values that describe such things as array size, various limits, special values, and the like.

Variables in Interfaces

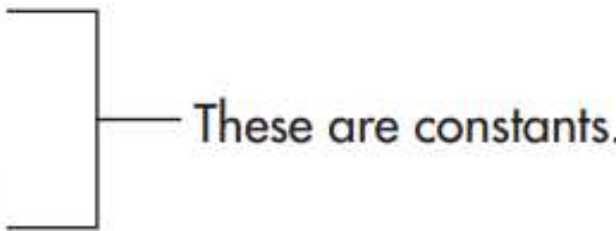
- To define a set of shared constants, create an interface that contains only these constants, without any methods.
- Each file that needs access to the constants simply “implements” the interface.
- This brings the constants into view. Here is an example:

Variables in Interfaces

```
// An interface that contains constants.
interface IConst {
    int MIN = 0;
    int MAX = 10;
    String ERRORMSG = "Boundary Error";
}

class IConstD implements IConst {
    public static void main(String[] args) {
        int[] nums = new int[MAX];

        for(int i=MIN; i < 11; i++) {
            if(i >= MAX) System.out.println(ERRORMSG);
            else {
                nums[i] = i;
                System.out.print(nums[i] + " ");
            }
        }
    }
}
```



These are constants.

Interfaces Can Be Extended

- 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.


Interfaces Can Be Extended

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



B inherits A.

A diagram consisting of a horizontal line pointing left from the 'A' in 'B extends A' to a vertical line pointing down, which then turns left to point at the text 'B inherits A.'

```
// This class must implement all of A and B
```

```
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
}
```

Interfaces Can Be Extended

```
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();
    }
}
```