

Interfaces

- 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 their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface**.
- One class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined 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.
- 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. 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. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.
- **NOTE** Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.

Defining an Interface

- An interface is defined much like a class.
This is the 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 specifier 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.
- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods. 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 defined 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 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);
}
}
```
- Notice that **callback()** is declared using the **public** access specifier. **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.”
- **CAUTION** Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.
- 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.
- An interface reference variable only has knowledge of the methods declared by its **interface** declaration. Thus, `c` could not be used to access `nonInterfaceMeth()` since it is defined by **Client** but not **Callback**.
- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of
callback");
System.out.println("p squared is " + (p*p));
}
}
```

Now, try the following class:

```
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

- As you can see, the version of **callback()** that is called is determined by the type of object that **c** refers to at run time.

## Partial Implementations

- If a class includes an interface but does not fully implement the methods defined 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.
- 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");
 }
}

```

- 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.