Interfaces

Using the keyword **interface**, you can fully abstract a class interface from its implementation.
Specifies what a class must do, but not how to do 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**.
Also, one class can implement any number of interfaces.

Interfaces are designed to support dynamic method resolution at run time.

General Form

```
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;
}
```

- *access* is either **public** or not used

- *name* is the name of the interface, and can be any valid identifier

- the methods which are declared have no bodies. 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 with a constant value.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

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:

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

- 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**.

- The type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

```
interface Callback {
void callback(int param);
}

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

- 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

```
interface Callback {
void callback(int param);
}

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.");
}
}
```

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
```

```
interface Callback {
void callback(int param);
}

class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}

// 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));
}
}
```

```
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
```

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.

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