17 INTERFACES

EL OBJECTIVOS:

- >> Learn It
- >> Live It
- >> Love It

The interface is Java's answer to multiple inheritance. It is a Java type that defines *what* should be done, but *not how* to do it. Interfaces are perhaps most useful when designing the API for your program. In this topic, we'll find out how to define an interface, how to implement one, and how to use it in program design.

Let's Get One Thing Straight

Let's get one thing straight. Right off the bat. There are really two ways that people in Javaland use the term interface. One is conceptual and one is concrete.

People sometimes talk about a program's interface, or even a class' interface. Remember that API is an acronym for Application Programming Interface. When used this way, it means *the thing with which we interact*. It's what is exposed to us that we can work with. It is the visible boundary of a class or a program or a programming language's libraries. This is the conceptual version of the term interface. It means the public (I really mean non-private) methods that we can call to do something.

On the other hand, an interface is a Java programming language construct, similar to an abstract class, that allows you to specify zero or more method signatures without providing the implementation of those methods. Remember the implementation is the code block that does the work. Let's look at the difference.

```
public void printMessage(int numberOfTimes);
// a method declaration in an interface.
```

The preceding code is an example of an interface method declaration. Notice how the signature ends in a semi-colon. There is no code that does the actual work of the `printMessage()` method.

An interface will consist of method signatures that look like this; there can be no implementation at all. You do the implementation in a class. An implementation of the `printMessage()` method might look like this.

```
public void printMessage(int numberOfTimes) {
  for (int i = 0; i <= numberOfTimes; i++) {
    System.out.println("Current message number " + i);
  }
}
```

Or the implementation could be different. Maybe a different implementation uses a `while` loop, and has a different String message, or doesn't print the current iteration of the loop as represented by `i`.

Imagine an interface with some method signatures, and a class that will implement the interface.

Interface: I have 5 method signatures.

Class: I want to implement them.

Interface. Okay. But then you have to implement all of them. You are not allowed to say that you implement me without implementing every single one of my methods.

Class: It's a deal.

An interface is a contract. It is binding between the interface and the class that implements the interface.

But why in the world would you want to do that? A class can implement whatever methods it wants. Why not cut out the middleman, go ahead and implement the methods you want, and then forget about the interface altogether? Well, you could do that. But there are actually several reasons why interfaces are very cool.

If it helps, you can think of an electrical outlet.

An electrical outlet is a wonderful invention. It is really, really a cool thing, the electrical outlet. The interface for every electrical outlet is *exactly the same* (okay, they're different in the United States than other places, and sometimes you get three little holes to plug stuff into, and sometimes only two; work with me here. Geez). You know that you will be able to use the electricity you need for your laptop, PlayStation, hair drier, or electric dog polisher as long as each of them have standard cords that will plug into an outlet. We could imagine a hypothetical interface called Pluggable that means that it has a standard set of prongs that will plug into a standard outlet. A bowl does not implement the Pluggable interface. Can't plug it in. The book "The Complete William Shakespeare" doesn't implement the Pluggable interface. Can't plug it in. However, an e-book reader does implement the Pluggable interface. You could read "The Complete William Shakespeare" on e-book or in paper form. Same text. Different implementation.

So back to our story.

Some Reasons Why Interfaces Are Very Cool

1. **They encourage smart application design.** Say that you have an application like an e-commerce site. You have to access a database a lot for different reasons. On one occasion, you need to retrieve products and display the catalog; another time, you need to store customer information. Now, these operations aren't related much in terms of the domain. But they do the same thing: interact with the database. You might create an interface called DataAccessor that defines all of the operations that can be done with the database: select, insert, update, delete, and so forth. Then, you have one standard way that all of the programmers on your project can write to. All of your database access contracts, even if written by people working on unrelated parts of the app, will look the same.
2. **They promote readability.** I have seen enough people come and go in different jobs that I am a firm believer in doing what you can to make your code readable. Interfaces promote readability because readers of your code know what to expect of a class that implements an interface. Also, it gives readers a second, concise location to overview what the class does.
3. **They promote maintainability.** The reason that they promote maintainability is more complicated, and we'll get to it in a moment. In a nutshell, if a class implements an interface, you can use the interface type as the reference type of instances of that class. That means that later you can swap out the actual implementation without breaking any of your code. That ability is an instance of polymorphism, one of the pillars of object-oriented programming. More on this in a mo.
4. **They allow flexibility in your class definitions.** In Java, you are only allowed to explicitly extend, or inherit from, one class. This is different from languages like C++ and C# that allow you to extend multiple classes. By allowing Java classes to extend from one class and also implement an interface at the same time, we can in effect skirt the fact that we're not allowed multiple inheritance, because of polymorphism. In fact, Java programmers are often encouraged to implement an interface instead of inheriting from a class when faced with that choice. Programmers probably come to such a crossroads when creating a new thread. You can create a thread by extending the Thread class or by implementing the Runnable interface. They both get you a thread. But you can implement as many interfaces as you want—you can only inherit from one class. So the idea is to prefer interfaces over inheritance unless you really are extending the definition of what your superclass can do. That is, nine times out of ten we aren't actually extending the functionality of a thread when we write `extends Thread`. We're not making a more specific, more functional kind of thread; we're just wanting to spawn a new instance of a regular old thread so that we can execute some code separately from the main program thread. In this case, you should implement Runnable and get your thread that way. That leaves you with the ability to extend a different class if you want to, one whose functionality your class is closer to, or that you really need to inherit from.

How to Write an Interface

The Java API is full of terrific interface definitions. One commonly used interface is java.lang.Runnable, which can be implemented to create a thread. To write an interface, you can look at the many good examples in the Java API.

When writing your own, you just use the keyword `interface` instead of `class`, and then don't include the method implementations. Like this:

```
public interface MyInterface {
      public void someMethod(long someParam);
}
```

As you can see, you just use the `interface` keyword in place of `class`. And then you substitute a semi-colon for the curly braces holding the method implementation. That's all you have to do in a nutshell. In the following sections, we'll see the many twisting, cavernous corridors that lurk beneath the deceptively simple interface.

After you have an interface, you implement it. That means you declare a class that implements the interface, and then write the implementation for each method in the interface. And the method signatures must match exactly. Like this:

```
public MyClass implements MyInterface {
   //since I said "implements MyInterface",
   //I must include this method, or
   //this class won't compile
   public void someMethod(long someParam) {
       //this is my implementation.
   }
}
```

You can add methods and other stuff to your implementing class if you want to.

Interfaces Versus Abstract Classes

There are differences between an interface and an abstract class, though commonly the two are confused. Why do you need an interface? When should I use an interface and when an abstract class? Let's let the two duke it out for themselves, and you decide.

Monsters fight!

Round One: *An interface is a totally abstract class*. In an abstract class, you can define some methods that are abstract, and some methods that are fully implemented. Any class that extends that abstract class must implement the abstract methods, but it inherits the functionality as implemented in the abstract class. This is very cool if you need that kind of structure in your program, but can make your API a little snaky.

Round Two: In an interface, there can be no implementation whatsoever. You can say this:

```
List starWarsFigures = new ArrayList(500);
```

Then, your reference type is the interface type (List)! That means that later in your code you can change the implementation without breaking any of the code that does something with your starWarsFigures object. Like this:

```
List starWarsFigures = new Vector();
```

This is very cool if you need that kind of flexibility, which is often a good thing.

Round Three: An interface is less flexible in how its variables and methods are declared. Here are the rules about writing an interface.

Rules for Writing an Interface

- Declare an interface using the keyword `interface`.
- An interface may extend zero or more interfaces if it likes, but it cannot extend a class, because then it would inherit functionality, and interfaces cannot have functionality. They can only talk about it.
- Interface methods cannot be static.
- Interface methods are implicitly abstract. For that reason, you cannot mark them `final` (duh), `synchronized`, or `native` because all of these modifiers tell how you're going to implement the method, and you're voluntarily giving up that ability when you write the method in an interface.
- `strictfp` is okay on an interface. It is okay because you can evaluate compile-time constants using `strictfp` rules within an interface.
- All interface methods are implicitly abstract and public, *regardless of what you write in the interface definition!* It is true. The interface method declaration `void biteIt(int times)`, despite its apparent access level of default, actually has `public` access. Try it. If you write a class in another package beyond the visibility of default access, and include the seemingly legal implementation of `void biteIt(int times) { ; }`, the compiler will tell you that you cannot reduce the visibility of the method from `public` to default. They're all abstract; they're all `public`.
- An interface can define variables. But all variables defined in an interface must be declared `public`, `static`, and `final`. Many Java programmers have adopted the practice of defining only variables within an interface and putting constants in it. This works to get at shared constants, but it is a workaround and is no longer necessary if you're using J2SE SDK 5.0. It features a new static import facility that allows you to import constants just as you would a class or package.
- It should be obvious by now that an interface cannot implement another interface or a class.
- You may modify your methods using the keyword `abstract` if you like, but it will have no effect on compilation. Methods in an interface are already abstract, and the Java Language Specification says that its use in interfaces is obsolete.
- Likewise, the interface itself is already abstract. So you can do this if you want: `public abstract interface Chompable {}`. But there's no point; it's redundant.

- Interfaces have default access by default (!). So this is legal: `interface Chompable { }`. But if you want your interface to have public access, use that modifier in the interface declaration.
- You cannot declare an interface as having private access. It doesn't make any sense. No one could implement it. So `private interface Chompable { }` gets you a compiler error for your trouble.
- `public`, `static`, and `final` are implicit on all field declarations in an interface.

There are some weird things to keep in mind.

Interfaces can be declared `private` or `protected` if they are declared nested inside another class or interface. The following will compile, though its usefulness is dubious at best.

```
public class interface test {
    private interface myinterface{ }
}
```

Only an inner class of the class of which the interface is declared can implement the interface.

Is and Does

Remember that the job of an interface is to designate a role that a class will play in the society of the application. Whereas a class (especially an abstract class) defines what something is, an interface defines what it can do.

Which of the following would you make into an interface, and which would you make into an abstract class?

Person
Employee
Programmer
Skier
WildAnimal
DataAccessor
Swimmer
WestCoastChopper

You'll see a 3-dimensional pattern emerge if you relax your eyes and stare at it long enough. This sort of thing is important to keep in mind as you design your application. You can do all of your applications without ever using an interface. But again, they clarify your API and provide you with flexibility down the road.

Constants

Java programmers commonly define constants inside their interfaces, if it makes design sense. You can do so using variables in an interface because the values will be present instantly at runtime and their values shared among all classes implementing your interface, because they are `static` and `final`.

FRIDGE

Like the `public` and `abstract` deal, interface variables are implicitly `public`, `static`, and `final`. That is, the following are equivalent within an interface: `public static final String DB_NAME = "Squishy"` and `String DB_NAME = "Squishy"`. Likewise, you are not allowed to do this: `private String x;`

Here is how you do it.

```
public interface IDataAccessor {
    String DB_NAME = "Squishy";
}
```

Note that it only makes sense to define interface constants if the variables are tightly related to your interface definition. There is a pattern that has been popular among developers where they use interfaces for the sole purpose of defining constants, so you might come across a lot of code to that effect. But J2SE 5.0 introduces other mechanisms that are more appropriate for this sort of thing—namely typesafe enums and static imports.

The reason that you don't want to use this old pattern anymore (assuming you're using it) is that it confuses the API. To access the constants defined in the interface, you must include `implements InterfaceName` in your class definition—and that isn't really true. You aren't implementing any functionality defined by such an interface; you just want the variables. It's a workaround, and now there are language features to handle that situation, so you don't have to resort to it.

Interface Inheritance

Yes, an interface can extend another interface. Just say that it does, like this:

```
public interface ISpy extends
        IInternationalManOfMystery {...

}
```

This is rarely used in practice. It means just what you would expect: Now the class that implements ISpy must also implement all of the methods from IInternationalManOfMystery.

Also, an interface is allowed to extend more than one interface. Just separate them by commas. Notice that this is different than regular classes, which are not allowed to extend more than one class.

Implementing Interfaces

You have to do two things to implement an interface. You have to announce that your class will implement the interface in your class declaration, like this:

```
public class SuperHero implements IFly {...}
```

Then, you have to implement each method defined in an interface. Most IDEs, including Eclipse, will tell you the names of the methods you need to implement as a convenience. Note that whether you have implemented each method you're supposed to is checked at compile time, not at runtime.

Remember that the purpose of an interface is to say that the implementing class must have methods that match the signatures. That doesn't say anything about the quality of the implementation. Say we have an interface called IFly (which we'll define next), and it has one method `void accelerate(int speed);`. The following is legal, and will compile:

```
public class SuperHero implements IFly {
     public void accelerate(int i) {}
}
```

This can be convenient if an interface has a method or two that you don't really need. But if you find yourself doing this sort of thing with any frequency, you might take a second look at whether you really ought to be implementing that interface, and if there isn't some other way to solve the problem. If you are the designer of the interface, you might want to reexamine your interface design to make sure that you are defining what users (in this context, programmers who want to implement your interface, including you) really need. By the same token, if you find in your code a number of methods that programmers need to define over and over again, consider moving them into the interface and making them part of the contract. This could have the effect of tightening and strengthening your abs. I mean your API.

A class can implement more than one interface. Separate each interface with a comma in the class declaration, like this:

```
public class RockStar implements IGuitarPicker,
    ILeatherPantsWearer { }
```

Abstract Class Implementation of Interfaces

This is something very weird. Actually, I'll pose it as a question and see what you think. Previously, we had our IFly interface, and our regular old class SuperHero implemented it.

(Deep, oily announcer voice) For 50 points, can an abstract class implement an interface? *(After slight pause, with greater significance)* Can an abstract class implement an interface?

It is a naming convention that you'll often see to prefix "I" before the name of an interface in order to identify it easily as an interface. If you've programmed in VB, you're probably used to Hungarian notation, which serves a similar purpose.

Think about it (after all, it's worth 50 points). Can an abstract class implement *anything*? Actually, yes, it can. It can have both abstract and concrete methods. So the answer must be yes. Because you can do this:

```
public abstract class SuperHero implements IFly {
   public void accelerate(int speed) {
      // some code here to do accelerating...
   }
}
```

Any class that extends SuperHero will inherit this functionality, or alternatively, it can decide to override the `accelerate()` method if it wants to, and define a different implementation.

That was easy enough. But here's the killer: Is the following legal?

```
public abstract class SuperHero implements IFly { }
```

Notice that there is not only no mention of the `accelerate()` method, there is no code at all. How could an abstract class say it implements an interface that it doesn't implement? It must be wrong!

Actually[1], that code compiles without a problem. Because there is another contract at work when you define an abstract class, and that contract is that the interface methods must *have been implemented* by the first concrete subclass of the abstract class. The first concrete class is not required to implement all of them, just those that have not been previously implemented by some abstract class that implements the interface up the hierarchy. The following series of classes is legal and compiles.

IFly.java

```
public interface IFly {
     void accelerate(int speed);
     void slowDown(int speed);
}
```

Now an abstract class will say it implements the IFly interface.

SuperHero.java

```
public abstract class SuperHero implements IFly { }
```

JusticeLeagueSuperHero.java

```
public abstract class JusticeLeagueSuperHero
          extends SuperHero
{
     public void accelerate(int speed) {
       //some real implementation code!
```

[1] I hate it when people say, "actually." It sounds soooo snotty. As if there is some other life that you're living over there in happy fantasyland where everybody has all wrong ideas about stuff and you're so dumb that it can't be imagined how you even function.

```
        System.out.println("Accelerating to " + speed
        + "mph");
    }
}
```

Notice that JusticeLeagueSuperHero implements the `accelerate()` method, but not the `slowDown()` method, which is also required by the interface. The code compiles because it is also declared abstract, and it is expected that one day someone will want to make a concrete subclass of JusticeLeagueSuperHero, so that you can say `new SuperHero()` and get to work. Of course, if no one ever makes a concrete subclass that implements the `slowDown()` method, the interface is not really implemented, even though we said it was. But that couldn't matter less, because we can't ever make any objects.

So now let's look at the final class in our implementation of this interface, the concrete class SomeMoreSpecificSuperHero.

SomeMoreSpecificSuperHero.java

```
public class SomeMoreSpecificSuperHero
                extends JusticeLeagueSuperHero {

    public void slowDown(int speed) {
        //some real implementation code here!
        System.out.println("Slowing down
                to " + speed " mph");
    }
}
```


In our little hierarchy, we have an interface, an abstract class (SuperHero) that implements the interface but doesn't actually implement any methods, a second abstract class (JusticeLeagueSuperHero) that extends SuperHero and provides a concrete implementation of one method in the interface, and then our first concrete class (SomeMoreSpecificSuperHero) that extends JusticeLeagueSuperHero and implements the remaining method. By the time we get to the concrete class, everything has an implementation. Cool.

Well, I'm going to get a beer and make a pizza. Thanks for talking about interfaces with me. I'm feeling much better now.