What Are Java Interfaces?

What are interfaces in Java, and what good are they? What are they used for? In this article, we'll take a look at interfaces and their uses in Java.

What Do Interfaces Do?

An interface in Java is basically a way of specifying what methods a class should have. Since Java 8, it's also possible to specify default implementations for those methods.

An interface declaration looks a lot like a class declaration, except none of the methods contain code.

Let's take an example. In the following code I've declared an interface named **Animal**, which specifies what kind of methods a class should have if it "implements" this interface.

Animal.java:

```
public interface Animal {

    void eat();

    void call();

    void move(int x, int y);

}
```

This interface specifies that any class that "implements" this interface must have at least these three methods. It may have more methods, but it must have these three. The **move** method must take two parameters of type **int** and the other two methods must take no parameters. All the methods, in this particular case, return **void**.

It's not necessary to specify **public** here, because interface methods are public by default.

Now we can create a class that implements this interface.

Cat.java:

```
public class Cat implements Animal {


    @Override

    public void eat() {

        System.out.println("Nom nom nom");
    }

    @Override
    public void call() {
        System.out.println("Meouww!!");

    }

    @Override
    public void move(int x, int y) {
        System.out.println("Moving to " + x + ", " + y);

    }

}
```

Since we've written `public class Cat implements Animal`, the **Cat** class is forced to have at least the methods specified in the interface (but it may have others too). I've created dummy implementations for the methods here. But what use is this?

What Use Are Interfaces?

I can think of four main uses of interfaces. They are:

- Designing programs using interfaces
- Treating classes with groups of identical methods in the same kind of way
- Passing code to methods
- Defining constants

Coding to Interfaces

Some people like to start designing a program by creating the interfaces that the classes composing the program will implement. You first think about what methods each class will have, define the interfaces, and only then start actually implementing methods.

This is sometimes referred to as "coding to interfaces".

This is all very well, but can't we actually *do* something with interfaces in our code, rather than just using them to design stuff? The answer is yes.

Treating Similar Classes in a Similar Way

One thing interfaces allow us to do is to treat classes that implement the interface in a simliar way. Now that we've got an `Animal` interface and we've got a `Cat` class that implements it, we can do this:

Cat.java:

```
Animal myCat = new Cat();
```

On the left of the equals sign we declare a variable of type `Animal`, the interface type. On the right of the equals we create a new object of the type `Cat`, making the `Animal`-type variable point at the new `Cat` object.

This is a little bit limited, because even if we add more methods to `Cat`, the `Animal`-type variable will only let us directly call those methods that it knows about; the methods in the `Animal` interface.

But now suppose we create a new class that implements the interface.

Dog.java:

```
public class Dog implements Animal {


    @Override
    public void eat() {
        System.out.println("Chmp chmp chmp!");


    }


    @Override
    public void call() {
        System.out.println("Woof!!");


    }


    @Override
    public void move(int x, int y) {
        System.out.println("Checking out smell at " + x + "," + y);


    }
}
```

Now we can use the interface to store both types of class in an array, for example:

```
Animal[] animals = new Animal[4];


animals[0] = new Cat();
animals[1] = new Dog();
animals[2] = new Dog();
animals[3] = new Cat();
```

In the same way that we could use a variable of the interface type to refer to any object of a class that implements it, we can do the same sort of thing with arrays, ArrayLists, and so on.

Passing Code to Methods with Interfaces

Suppose we want to pass some code to a method. Why might we want to do this? For example, when we create Swing desktop apps with Java, we need to tell buttons what code they should run. We can do this because the class that creates buttons in Swing, **JButton**, has a method intended for exactly that purpose; it's called **addActionListener**.

But how do we pass code to this method? **addActionListener** accepts a parameter of an interface type; the **ActionListener** interface. This interface has one method called **actionPerformed**.

So the button knows that it should call the **actionPerformed** method of the object that's passed to **addActionListener** when it's clicked.

```
class ButtonResponse implements ActionListener {


    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("This runs when the button is clicked.");
    }


}


JButton button = new JButton();
```

```
button.addActionListener(new ButtonResponse());
```

We might also choose to use anonymous class syntax instead of declaring a class here. It's possible to implement an interface "on the fly", if we only need one object of the class. This is called creating an anonymous class. Here's a complete example.

Test.java:

```
interface Runner {
    public void runMe();
}


public class Test {


    public static void main(String[] args) {


        runSomething(new Runner() {


            @Override
            public void runMe() {
                System.out.println("Running!");
            }


        });




    }


    public static void runSomething(Runner theRunner) {
        theRunner.runMe();
    }
 }
```

Code output: `Running!`

It looks a little strange, but it works.

We can do exactly the same thing with a lambda expression in Java 8, with exactly the same effect:

Test.java:

```
interface Runner {
    public void runMe();
}

public class Test {

    public static void main(String[] args) {

        runSomething(()->{ System.out.println("Running!");});



    }

    public static void runSomething(Runner theRunner) {
        theRunner.runMe();
    }
 }
```

Code output: `Running!`

Defining Constant in Interfaces

Another use of interfaces is that we can define constants in interfaces. While it's often better to use an `enum` and this practice is frowned upon by many, it's nevertheless commonly done in Java.

If we create this interface:

```
interface Constants {
    int TOTAL_ANIMALS = 50;
}
```

Then we can write, for example:

```
System.out.println(Constants.TOTAL_ANIMALS);
```

Note that constants in interfaces are **public**, **final** and **static** by default.

Conclusion

While interfaces initially appear quite useless, they are in fact a powerful and useful construct in the Java language. Once you know how to use them, you'll probably use them a lot.