Interfaces in Java

Introduction

In this article from my free Java 8 course, I will explain the topic of interfaces in Java.

Definition of the Interface

Interface is a generic term that is used widely across fields. Wiktionary defines it as “The point of interconnection between entities”. This term has been adapted to programming and plays a key role in Object Oriented Programming (OOP). In OOP, an Interface is defined as “a piece of code, defining a set of operations that other code must implement”.

History of Interface

Even though many believe that Java was where the idea of an *Interface* was initially introduced to programming, it actually was introduced by Brad Cox and Tom Love, the creators of Objective C, with the concept of the protocol.

The reason Java developed interfaces was to improve upon inheritance in C++. You could say that Java was the successor to C++ and in C++ they used a model involving multiple inheritance. Multiple inheritance is more complicated and problematic than the single inheritance model that was used in Java. It also is more difficult to implement multiple inheritance in a compiler. Interfaces were introduced as a substitute for multiple inheritance. It seems like, funnily enough, that interfaces weren’t introduced into Java to create “cleaner, more modular, and clearly separated code”, but rather just to compensate for the fact that Java didn’t support multiple inheritance. Nowadays however, that’s exactly what interfaces are useful for.

Writing an Interface

In this example we’re going to create a *CarService* class. We know that we want our *CarService* class to have a method called drive. When *drive* is invoked in the *CarService* class, we’re going to cause all the cars in our service to drive.

```
package com.marcusbiel.java8course;

public class CarService {

     public void drive() {
```

```
            BMW bmw = new BMW();
            Porsche porsche = new Porsche();
            Mercedes mercedes = new Mercedes();
            bmw.drive();
            porsche.drive();
            mercedes.drive();
      }
}
```

Example 1

As you can see in Example 1, our *CarService* creates three more objects. At this moment, the *CarService* class is deeply coupled to these other classes of cars. Each car is its own class, with no real connection besides the fact that they each have a *drive method*. This is something we don't want and I'll explain why in more detail later.

Here is where we can improve our code using an *interface*. In our code we have three different cars. All of these cars can drive, and you can already see we have three different drive methods. We could just create a *Car* class and a *carType* value in that class, but the drive method in this hypothetical class will force all our different types to drive the same way. This is an issue, because different cars drive in different ways. For example, the specific car types, (BMW, Mercedes, and Porsche), all have their own unique engines. On the other hand we've already recognized that all these cars drive and they also have other similarities like four wheels and two axles. These common features can be grouped together and accessed without knowledge of the particular car type using an *Interface*.

```
package com.marcusbiel.java8course;

public interface Car {
      void drive();
}
```

Example 2

Now we have defined an *Interface* called Car which contains the declaration for the *drive* method. Please note, by default all methods in an *interface* are "public abstract", so we don't need to include those modifiers in our methods and one shouldn't do it because it would just clutter the code. An abstract method requires any class that implements this interface to provide concrete implementation of the abstract method. Similarly, all class level variables declared in an *interface* have the default modifiers "public static final". Typically, while you can, you don't want to include constants in an interface. If you create a constant called "MAX_SPEED" at an interface level, you are adding concrete values to an interface. The goal of interfaces is to be 'lightweight', without any implementations. Like a contract or a blueprint, interfaces define 'what', but not 'how'. These implementation details should be put within a class or even better in an enum.

Subclass Implementing Interface

Let's modify our BMW class so that it "implements" the *Car* interface. This defines the BMW as a type of *Car* and it will adhere to the contract, or role, specified by Car. It is mandatory in any class that implements an interface to *override* all the abstract methods of the interface. In this case, BMW must override the *drive* method. We're also going to implement the *Loggable, Asset,* and *Property* interfaces. To implement multiple interfaces, separate each interface name with a comma.

```
package com.marcusbiel.java8course;

public class BMW implements Car, Loggable, Asset, Property {

    public void drive() {
        System.out.println("BMW driving...");
    }

    public int value() {
        return 80000;
    }

    public String owner() {
        return "Marcus";
    }

    public String message() {
        return "I am the car of Marcus";
    }
}
```

Example 3

For each of the interfaces I implemented, I had to override a method as I did above. Now we can revisit our *CarService* class. Assuming we implemented *Car* for the BMW, the Mercedes, and the Porsche, we can clean up our code. The first thing we can do is instantiate all of our Car types as Car:

```
Car bmw = new BMW();
Car porsche = new Porsche();
Car mercedes = new Mercedes();
```

Example 4

The objects are still all their specific types, however we are referencing with an interface to the object. By doing this our reference variable will only allow us to use methods provided by the given interface, and now the object plays a certain “role” in the given context. We also implement other interfaces Loggable, Asset and Property that would only allow us to use different sets of methods. These ‘lenses’ that our reference variable could act as allows a BMW object to fulfill different roles in different contexts.

Now we can use ‘Car’ for all three of the cars and it would make our code much more flexible. Even if we added new types of cars that implement the *Car* interface at a later time, we can still deal with them without even knowing that they would exist when we wrote *CarService*. Also, because all of our cars are actually implementing the *Car* interface, we can clean our code up even more using a foreach loop. We can also now retrieve all of our cars from a database, because no matter the type of Car, they all can be stored in one Array.

```
package com.marcusbiel.java8course;

public class CarService {
     public void drive() {

     [...] //dynamically retrieving cars from a database

          for (Car car : cars) {
               car.drive();
           }
     }
}
```

Example 5

Now our *CarService* doesn’t have any specific implementations anymore. This is called “decoupling”. The *CarService* class only uses and knows about the *Car* interface, not any specific types of cars.

Pros and Cons of Interface

To conclude, I’m going to discuss the pros and cons of interfaces in Java.

As always, if I’m adding more code, I have to actually type that code down and create files which makes the code slightly more complex. I also have to implement this method in my other classes which could add some complexity. While these may not seem like major problems in a small project, like this example, if used in the wrong context, the overhead will become an issue. So I would discourage using interface simply for the sake of it.

On the other hand, Interface plays an important role in having decoupled Java code. Declaring a reference variable of type interface allows you to substitute for different car types ***at runtime***. For example our *CarService* class can deal with either a *BMW* or a *Mercedes* based on different scenarios.

In practice, big teams can use this feature very powerfully. *Interface* defines the contract and different sub teams can work on different classes independently while sticking to the same basic guidelines provided by the interface. To reiterate, you shouldn't use interface just for the sake of having it, but when you use it properly it is one of the most powerful tools Java has to offer.

Thanks for reading!