Interfaces revisited (Java 8)

1 Introduction

Java 8 added a feature to `interface` that makes a significant change to its capabilities. Prior to Java 8, an interface could not define any type of implementation (other than constants). An interface could define only "what", but not "how". Java 8 changes this significantly. It is possible to add a *default* implementation to an interface method. However, default methods constitute what is, in essence, a special-use feature, and the original intent behind interface still remains (although they do effect the interpretation of abstract classes as there is now an overlap in meaning).

As a general rule, you will still create and use interfaces in which no default methods exist. For all previous versions of Java, the methods specified by an interface were abstract, containing no body. A *default method* lets you define a default implementation for an interface method. Therefore, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used (you will also see this term used in the language C#).

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would *break* existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided; the addition of a default method will not cause pre-existing code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called `remove()`, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and non-modifiable sequences, then `remove()` is essentially optional because it wont be used by non-modifiable sequences. In the past, a class that implemented a non-modifiable sequence would have had to define an empty implementation of `remove()`, even though it was not needed. Now a default implementation for `remove()` can be specified in the interface that does nothing

(or throws an exception). Providing this default prevents a class used for non-modifiable sequences from having to define its own, placeholder version of `remove()`. Thus, by providing a default, the interface makes the implementation of `remove()` by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of `interface`: that it does not maintain state information. Therefore the main difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with Java 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify what and not how. However, the inclusion of the default method gives you added flexibility.

2 The basics

An interface default method is defined similar to the way a method is defined by a class. The primary difference is that the declaration is preceded by the keyword default. For example, consider this simple interface:

```
package basics;

public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
}
```

`MyIF` declares two methods. The first, `getNumber()`, is a standard interface method declaration. It defines no implementation whatsoever. The second method is `getString()`, and it does include a default implementation. In this case, it simply returns the string `"Default String"`. Pay special attention to the way `getString()` is declared. Its declaration is preceded by the `default` modifier. This syntax can be generalised. To define a default method, precede its declaration with the keyword `default`.

As `getString()` includes a default implementation, it is not necessary for an implementing class to override it. If an implementing class does not provide its own implementation, the `default` is used. For example, the `MyIFImpl` class shown below is perfectly valid:

```
package basics;

//Implement MyIF.
class MyIFImpl implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

The following code creates an instance of `MyIFImpl` and uses it to call both `getNumber()` and `getString()`.

```
package basics;

//Use the default method.
class DefaultMethodDemo {
  public static void main(String args[]) {

    MyIF obj = new MyIFImpl();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
  }
}
```

The output is shown here:

```
100
Default String
```

As you can see, the default implementation of `getString()` was automatically used. It was not necessary for `MyIFImpl` to define it. Thus, for `getString()`, implementation by a class is optional. (Of course, its implementation by a class will be required if the class uses `getString()` for some purpose beyond that supported by its `default`.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, `MyIFImpTwo` overrides `getString()`:

```
package basics;

class MyIFImplTwo implements MyIF {
  // Here, implementations for both getNumber( ) and getString( ) are
  // provided.
  public int getNumber() {
    return 100;
  }

  public String getString() {
    return "This is a different string.";
  }
}
```

Now, when `getString()` is called, a different string is returned.

3 Example

Let us consider an interface `IntStack`, and for the sake of discussion, assume that `IntStack` is widely used and many programs rely on it. Further assume that we now want to add a method to `IntStack` that *clears* the stack, enabling the stack to be re-used. Therefore, we want to evolve the `IntStack` interface so that it defines new functionality, but we dont want to break any preexisting code. In the past, this would have been impossible, but with the inclusion of default methods, it is now easy to do. For example, the `IntStack` interface can be enhanced like this:

```
package stack;

public interface IntStack {
  void push(int item); // store an item

  int pop(); // retrieve an item

  // Because clear() has a default, it need not be
  // implemented by an existing class that uses IntStack.
  default void clear() {
    System.out.println("clear() not implemented.");
  }
}
```

Here, the default behaviour of `clear()` simply displays a message indicating that it is not implemented. This is acceptable because no pre-existing class that implements `IntStack` would ever call `clear()` because it was not defined by the earlier version of `IntStack`. However, `clear()` can be implemented by a new class that implements `IntStack`. Furthermore,

`clear()` needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

In real-world code, `clear()` would have thrown an exception, rather than displaying an error message. You might want to try modifying `clear()` so that its default implementation throws an `UnsupportedOperationException`.

4 Implications of `default` for multiple inheritance of implementations

Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance of implementation. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behaviour is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behaviour. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called `Alpha` and `Beta` are implemented by a class called `MyClass`. What happens if both `Alpha` and `Beta` provide a method called `reset()` for which both declare a default implementation?

- Is the version provided by `Alpha` or the version by `Beta` used by `MyClass`?

Consider a situation in which `Beta` extends `Alpha`.

- Which version of the default method is used?
- What if `MyClass` provides its own implementation of the method?

To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

1. In all cases, a class implementation takes priority over an interface default implementation.
   Thus, if `MyClass` provides an override of the `reset()` default method, `MyClass` version is used.
   This is the case even if `MyClass` implements both `Alpha` and `Beta`. In this case, both defaults are overridden by `MyClass` implementation.

2. In cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.
   Continuing with the example, if `MyClass` implements both `Alpha` and `Beta`, but does not override `reset()`, then an error will occur.

3. In cases in which one interface inherits another, with both defining a common default method, the inheriting interfaces version of the method takes precedence.
   Therefore, continuing the example, if `Beta` extends `Alpha`, then `Beta`s version of `reset()` will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of `super`. Its general form is shown here:

```
InterfaceName.super.methodName()
```

For example, if `Beta` wants to refer to `Alpha`s default for `reset()`, it can use this statement:

```
Alpha.super.reset();
```

5 Use static methods in an interface

Java 8 added another new capability to interface: the ability to define one or more static methods. Like static methods in a class, a static method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method. Instead, a static method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

```
InterfaceName.staticMethodName
```

Notice that this is similar to the way that a static method in a class is called.

The following shows an example of a static method in an interface by adding one to `MyIF`, from the earlier section. The static method is `getDefaultNumber()` — it simply returns zero.

```
package stat;

public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }

  // This is a static interface method.
  static int getDefaultNumber() {
    return 0;
  }
}
```

The `getDefaultNumber()` method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of `MyIF` is required to call `getDefaultNumber()` because it is `static`. Static interface methods are not inherited by either an implementing class or a sub-interface.