Java 8 Default Methods Explained in 5 minutes

Ernest
Duodu

In my previous articles, we have looked at Java 8 Lambda Expressions and Streams. In this article will be looking at Defaults Methods which is another cool feature of Java 8.

Default methods enable us to add new functionalities to interfaces without breaking the classes that implements that interface. Lets take a look at the example below.

```
public class MyClass implements InterfaceA {

    /**
     * @param args the command line
arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Override
    public void saySomething() {
        System.out.println("Hello World");
    }

}

interface InterfaceA {
  public void saySomething();

}
```

The code above shows class `MyClass` implementing InterfaceA's method `saySomething()`. Now lets add a new method called `sayHi()` to InterfaceA. By doing so, we have introduce a problem to class `MyClass` as it will not compile until we provide implementation for method `sayHi()`.
This is when Defaults methods becomes useful. By Adding the keyword ***default*** before the method's access modifier, we do not have to provide implementation for the method `sayHi()` in class `MyClass`.

In *'the strictest sense'*, Default methods are a step backwards because they allow you to 'pollute' your interfaces with code. But they provide the most elegant and practical way to allow backwards compatibility. It made it much easier for Oracle to update all the Collections classes and for you to retrofit your existing code for Lambda.

```
public class MyClass implements InterfaceA {

    /**
     * @param args the command line
arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Override
    public void saySomething() {
        System.out.println("Hello World");
    }

}

interface InterfaceA {

    public void saySomething();

    default public void sayHi() {
      System.out.println("Hi");
    }

}
```

Note that we have to provide implementation for all ***default*** methods. So ***default*** methods provides us the flexibility to allow methods to be implemented in interfaces. The implementation will be used as default if a concrete class does not provide implementation for that method.

Conflicts with Multiple Interface.

Since classes in java can implement multiple interfaces, there could be a situation where 2 or more interfaces has a ***default*** method with the same signature hence causing conflicts as java will not know what methods to use at a time. This will then result in a compilation error with the message `MyClass inherits unrelated defaults for sayHi() from types InterfaceA and InterfaceB`
Lets take a look at the example below.

```
public class MyClass implements InterfaceA, InterfaceB
{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Override
    public void saySomething() {
        System.out.println("Hello World");
    }

}

interface InterfaceA {

    public void saySomething();

    default public void sayHi() {
        System.out.println("Hi from InterfaceA");
    }

}

interface InterfaceB {
     default public void sayHi() {
        System.out.println("Hi from InterfaceB");
    }
}
```

In order to work around situations like this,We will have to provide implementation for `sayHi()` method in the class `MyClass` therefore overriding both methods in `InterfaceA` and `InterfaceB`.

```
public class MyClass implements InterfaceA, InterfaceB {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Override
    public void saySomething() {
        System.out.println("Hello World");
    }

    @Override
    public void sayHi() {
        System.out.println("implemetation of sayHi() in
MyClass");
    }

}

interface InterfaceA {

    public void saySomething();

    default public void sayHi() {
        System.out.println("Hi from InterfaceA");
    }

}

interface InterfaceB {
     default public void sayHi() {
        System.out.println("Hi from InterfaceB");
    }
}
```

If we want to specifically invoke one of the `sayHi()` methods in either InterfaceA or InterfaceB, we can also do as follows:

```
public class MyClass implements InterfaceA, InterfaceB
{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    @Override
    public void saySomething() {
        System.out.println("Hello World");
    }

    @Override
    public void sayHi() {
       InterfaceA.super.sayHi();
    }

}

interface InterfaceA {

    public void saySomething();

    default public void sayHi() {
        System.out.println("Hi from InterfaceA");
    }

}

interface InterfaceB {
     default public void sayHi() {
        System.out.println("Hi from InterfaceB");
    }
}
```

Hopefully you have found this quick guide useful. Next Time I'll be looking at Java 8 Method References.

If you're a first-time reader, or simply want to be notified when we post new articles and updates, you can keep up to date by social media (Twitter, Facebook and Google+) or the Blog RSS.

-