Java 8 Interface Changes – static methods, default methods, functional Interfaces

One of the biggest design change in Java 8 is with the concept of interfaces. Prior to Java 7, we could have only method declarations in the interfaces. But from Java 8, we can have **default methods** and **static methods** in the interfaces.

Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes. As interface grows old, the number of classes implementing it might grow to an extent that it's not possible to extend interfaces. That's why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application.

Let's look into the default and static interface methods and the reasoning of their introduction.

Interface Default Method

For creating a default method in the interface, we need to use "**default**" keyword with the method signature. For example,

Interface1.java

```java
package com.journaldev.java8.defaultmethod;

public interface Interface1 {

    void method1(String str);

    default void log(String str){
        System.out.println("I1 logging::"+str);
        print(str);
    }
}
```

Notice that log(String str) is the default method in the `Interface1`. Now when a class will implement Interface1, it is not mandatory to provide implementation for default methods. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation.

Let's say we have another interface with following methods:

Interface2.java

```java
package com.journaldev.java8.defaultmethod;
```

```java

public interface Interface2 {

    void method2();

    default void log(String str){
        System.out.println("I2 logging::"+str);
    }

}
```

We know that Java doesn't allow us to extend multiple classes because it will result in the "Diamond Problem" where compiler can't decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too. Because if a class is implementing both `Interface1` and `Interface2` and doesn't implement the common default method, compiler can't decide which one to chose.

Extending multiple interfaces are an integral part of Java, you will find it in the core java classes as well as in most of the enterprise application and frameworks. So to make sure, this problem won't occur in interfaces, it's made mandatory to provide implementation for common default methods. So if a class is implementing both the above interfaces, it will have to provide implementation for `log()` method otherwise compiler will throw error.

A simple class that is implementing both `Interface1` and `Interface2` will be:

MyClass.java

```java
package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

    @Override
    public void method2() {
    }

    @Override
    public void method1(String str) {
    }

    @Override
    public void log(String str){
        System.out.println("MyClass logging::"+str);
        Interface1.print("abc");
    }
}
```

Important points about interface default methods:

1. Default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Default methods has bridge down the differences between interfaces and abstract classes.
3. Default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Default methods will help us in removing base implementation classes, we can provide default

implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from `java.lang.Object`. The reasoning is very simple, it's because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That's why to avoid confusion, we can't have default methods that are overriding Object class methods.
7. Default methods are also referred to as Defender Methods or Virtual extension methods.

Interface static methods

Static methods are similar to default methods except that we can't override them in the implementation classes. This feature helps us in avoiding undesired results incase of poor implementation in child classes. Let's look into this with a simple example.

MyData.java

```java
package com.journaldev.java8.staticmethod;

public interface MyData {

    default void print(String str) {
        if (!isNull(str))
            System.out.println("MyData Print::" + str);
    }

    static boolean isNull(String str) {
        System.out.println("Interface Null Check");

        return str == null ? true : "".equals(str) ? true : false;
    }
}
```

Now let's see an implementation class that is having isNull() method with poor implementation.

MyDataImpl.java

```java
package com.journaldev.java8.staticmethod;

public class MyDataImpl implements MyData {

    public boolean isNull(String str) {
        System.out.println("Impl Null Check");

        return str == null ? true : false;
    }

    public static void main(String args[]){
        MyDataImpl obj = new MyDataImpl();
        obj.print("");
        obj.isNull("abc");
    }
}
```

Note that `isNull(String str)` is a simple class method, it's not overriding the interface method. For example, if we will add @Override annotation to the isNull() method, it will result in compiler error.

Now when we will run the application, we get following output.

```
Interface Null Check
Impl Null Check
```

If we make the interface method from static to default, we will get following output.

```
Impl Null Check
MyData Print::
Impl Null Check
```

The static methods are visible to interface methods only, if we remove the isNull() method from the `MyDataImpl` class, we won't be able to use it for the `MyDataImpl` object. However like other static methods, we can use interface static methods using class name. For example, a valid statement will be:

```java
boolean result = MyData.isNull("abc");
```

Important points about interface static methods:

1. Interface static methods are part of interface, we can't use it for implementation class objects.
2. Interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can't define static methods for Object class methods, we will get compiler error as "This static method cannot hide the instance method from Object". This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.
5. We can use static interface methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

Functional Interfaces

Before I conclude the post, I would like to provide a brief introduction to Functional interfaces. An interface with exactly one abstract method is known as Functional Interface.

A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It's optional but good practice to use it.

Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use **lambda expressions** to instantiate them. A new package `java.util.function` with bunch of functional interfaces are added to provide target types for lambda expressions and method references. We will look into functional interfaces and lambda expressions in the future posts.