## New Rules for Interfaces

Java 8 changes the rules for interfaces. It adds the following:

1. interfaces can contain default implementations (code) for methods!

2. interfaces can contain static methods with code.

3. functional interfaces using the @Functional annotation

## Default Methods

Before Java 8 **all** interface methods were **`abstract`** (no method body). In Java 8, you can supply a "`default`" implementation for methods in an interface.

Suppose we have an interface for Money named **`Valuable`**. The Valuable interface has two methods: getValue() and getCurrency(). In Java 7 we would write:

```
public interface Valuable {
    double getValue();
    String getCurrency( );
}
```

In Java 8, we could specify default code for getCurrency that simply returns "Baht":

```
public interface Valuable {
    double getValue();
    default String getCurrency( ) { return "Baht"; }
}
```

To make the default getCurrency more general, you can add code to get the currency for the user's current Locale setting:

```
import java.util.*;  // for Currency and Locale
public interface Valuable {
    double getValue();  // abstract method

    default String getCurrency( ) {
        Locale locale = Locale.getDefault();
        return Currency.getInstance(locale).getDisplayName();
    }
}
```

Any code that "`implements Valuable`" can either override the `getCurrency()` method, or do nothing and use the default implementation.

## Static Methods

Java 8 interfaces can define static methods, including code. Any class that implements the interface will get the static method, as if the static method was defined in the class itself.

```
public interface VAT {
    static double VAT_RATE = 0.07;  // automatically "public final"
    static double getTax(Valuable v) {
         return v.getValue() * VAT_RATE;
    }
}
```

## Functional Interfaces

An interface with only one abstract method is called a "*functional Interface*", since they can be used like functions. Functional interfaces can be implemented as lambda expressions and method references. A lambda expression defines just one method, so the implicit type of a lambda (the target type) must be an interface with only one abstract method.

Similarly, a function reference refers to just one function. So, you can use a function reference in places that expect an interface with just one abstract method.

Some older interfaces (before Java 8) that qualify as functional interfaces are:

| | |
|---|---|
| *Comparable*<T> | `int compareTo( T other )` |
| *Comparator*<T> | `int compare(T a, T b)` |
| *Runnable* | `void run( )` |
| *Callable*<T> | `T call( )` |

Java 8 has many new functional interfaces in the package java.util.function. Most of them are special cases of one of these:

| Interface | Abstract Method | Purpose |
|---|---|---|
| ***Consumer<T>*** | `void accept(T arg)` | A function of one variable that doesn't return anything. It *consumes* the argument. |
| ***Supplier<T>*** | `T get( )` | Produces or "supplies" an object of type T, one object per call. |
| ***Predicate<T>*** | `boolean test(T arg)` | Performs a test on the argument. Used to build filters. |
| ***Function<T, R>*** | `R apply(T arg)` | A function of one parameter that produces a result. Can be used to *map* one kind of object to another. |
| ***BiFunction<T,U,R>*** | `R apply(T a, U b)` | Function of two parameters. |
| ***UnaryOperator<T>*** | `T apply(T arg)` | A unary operator. This is the same as *Function<T,T>* |
| ***BinaryOperator<T>*** | `T apply(T a, T b)` | A binary operator. Same as *BiFunction<T,T,T>* |

Many of these interfaces also have *default methods*. The default methods are used to "build" more complex functions.

For example, suppose we want a Predicate to test if a Double is greater than zero. Using a Lambda:

```
Predicate<Double> isPositive = (d) -> (d > 0.0);
```

You can test this predicate by invoking test( ) with some doubles:

```
isPositive.test( 2.5 )      // returns true
isPositive.test( 0.0 )      // returns false
```

We can create a new Predicate that tests for (d <= 0.0) by calling the **`negate()`** default method of Predicate:

```
Predicate<Double> notPositive = isPositive.negate( );
```

And test it:

```
notPositive.test( 0.0 )        // returns true
```

The *Consumer, Supplier, Predicate,* and *Function* interfaces all have type parameters. To make it possible to write Lambda expressions using primitive data types, Java 8 also adds many functional interfaces for primitive types like int and double (some people call this *interface pollution*). For example, for *Consumer* there are the following extra interfaces:

| | | |
|---|---|---|
| ***IntConsumer*** | `void accept( int x )` | Consumes an int |
| ***DoubleConsumer*** | `void accept( double x )` | Consumes a double |
| ***LongConsumer*** | `void accept( long x )` | Consumes a long |

Similarly for *Supplier* and *Predicate*. For *Function*, there are many specialized variations such as IntFunction, IntToDoubleFunction, IntToLongFunction, etc.

The *Functional Interfaces* serve two purposes:

1) provide convenient interface types for writing commonly used lambdas

2) provide interfaces used in the new *streams API*.

## Example using Functional Interfaces

**Student**

name: String
id: String
birthday: LocalDate

Suppose we have a Student class. A Student has an id, name, and birthday.

We want to print all the students born this month (so we can send them a birthday

A simple code for this is:

```
public void filterAndPrint( List<Student> students, int month ) {
      for(Student s : students ) {
            if (s.getBirthday().getMonthValue() == month)
                  System.out.println( s );
      }
}
```

In this code there is a test (a Predicate) and a Consumer. To make our code more general, let's rewrite the method so it accepts a Predicate (the test) and a Consumer (the action to perform).

```
public void filterAndDo( List<Student> students,
                         Predicate<Student> tester,
                         Consumer<Student> consumer ) {
        for(Student s: students) if (tester.test(s)) consumer.accept(s);
}
```

And use this new method to print students with birthday in May:

```
Month month = Month.May;  // an enum of the Months, used by LocalDate
// Test:  test the birthday month
Predicate<Student> hasBirthMonth =
       (s) -> s.getBirthday().getMonthValue() == month;
// Consumer:  print the student name and birthday
Consumer<Student> printBirthday =
       (s) -> System.out.println(s+" has birthday on "+s.getBirthday());

filterAndDo( students, hasBirthMonth, printBirthday );
```

We can use the new *Streaming interface* of collections instead of the for loop. In this case, we really don't need the method at all. We can just write:

```
students.stream().filter( hasBirthMonth ).forEach( printBirthday );
```

## Defining a Functional Interface

To define your own functional interface, prefix your interface declaration with @FunctionalInterface. However, any interface with exactly one abstract method can be used as a target type of a lambda expression even if you don't use this annotation.

## References

- In the Java API docs, the package desciption for java.util.function has a long description of the functional interfaces. The Java tutorial on Lambda expressions uses several function interfaces.
- "Enhancements in Java SE 8" online at https://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html