# Method Enhancements

## Overview

In this lab you'll refactor some Java applications to take advantage of the various method enhancement features available in modern Java.

## Modules for this lab (in the ModernJavaDev project)

* student.methods
* solution.methods

## Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Using method references to implement a functional interface
2. Defining default methods in an interface
3. Implementing a factory mechanism based on constructor references
4. (If Time Permits) Defining static methods in an interface

## Exercise 1: Using method references to implement a functional interface

In the Student.methodsmodule, expand the student.methods.functions package. Take a look at the following Java files:

* UnaryFunc.java  
  This is a simple generic interface that has a single method named perform(). This method represents the idea of a unary function, i.e. a function that takes a single parameter, performs an operation on it, and returns a result.
* Square.java and Cube.java  
  These classes implement UnaryFunc<Double> to perform different kinds of operation. Square performs a mathematical square operation, and Cube performs a mathematical cube operation.
* Main.java  
  This is the entry point for the program. The main() method calls doOp() several times, passing in two parameters each time:
  + The 1st parameter is an operand, i.e. the value to do some operation on.
  + The 2nd parameter is a UnaryFunc<T> object, which represents the specific operation to perform.

Inside doOp(), we invoke perform() on the UnaryFunc<T> object. Depending on what type of object this actually is, this will either cause the square or the cube to be obtained.

In Java 7, you'd define a separate class to represent every different type of operation. We've just defined two so far (Square and Cube), but the situation would quickly get out of hand if you wanted to represent lots of different operations. Java 8 onwards offers two simpler alternatives:

* Define a lambda expression to represent an inline implementation of the interface. This approach is appropriate if you have a one-off algorithm in just one place in your code.
* Use a method reference to point to an existing method in a class. This approach is appropriate if you want to reuse an algorithm in multiple places in your code.

Take a look at the following 2 classes, which provide several existing useful methods:

* DistanceConverter.java   
  This class has a couple of methods that convert distances from km to miles, and vice versa. Each method can fulfil the role of the perform() method in the UnaryFunc<T> interface, because they have the same signature.
* WeightConverter.java   
  This class has a couple of methods that convert weights from kg to pounds, and vice versa. Both of these methods have the same signature as the perform() method too.

Bearing all this in mind, extend main() as follows (use method references as appropriate):

* Call doOp() a couple of times, to convert a distance from km to miles and vice versa.
* Call doOp() a couple of times, to convert a weight from kg to pounds and vice versa.

**Exercise 2: Defining default methods in an interface**

Modern Java allows you to define default methods in an interface, as we discussed during the chapter. There are two main reasons for default methods in an interface:

* It allows you to add new methods to existing interfaces, without breaking any existing implementation classes (which obviously don't implement these new methods yet).
* It allows you to define an interface with lots of methods, of which all-but-one have a default implementation. As we’ve said several times now, this is what we mean by "functional interfaces". The notion of functional interfaces is crucial, because you can only use lambda expressions and/or method references in conjunction with functional interfaces.

With this in mind, add a default method to the UnaryFunc<T> interface. The method should be named formatResult, and should take a T as a parameter and return a String. The default method implementation should return a string with a format such as the following:

"Result: *parameter\_value*"

Then modify doOp() in Main.java, so that it uses the UnaryFunc<T> object's formatResult() method to format the result. Display the string result on the console, as at present.

Note what you just did. The doOp() method can invoke formatResult() on the UnaryFunc<T> object, without having to modify every single implementation class to implement that method. Of course, the implementation classes *can* define a formatResult() method if desirable. For example, implement formatResult() in Square and Cube and re-run the program.

## Exercise 3: Implementing a factory mechanism based on constructor references

In the Student.methodsmodule, expand the student.methods.factory package. The package contains 4 Java files:

* Logger.java  
  This is a simple interface with a single method named log().
* BriefLogger.java  
  This class implements the Logger interface to display brief log messages on the console. The idea is that client code can create a BriefLogger object at the start of a series of tasks, and then call log() several times to display a series of messages during these tasks.
* VerboseLogger.java  
  This class is similar to BriefLogger, except that it displays verbose messages (rather than brief messages) on the console.
* Main.java  
  This is the entry point for the program. The main() method calls doSomeStuff() several times. Each time it's called, doSomeStuff() creates a BriefLogger or a VerboseLogger (depending on the string parameter), and writes a series of log messages to the console. Run the program to verify you understand how it works.

doSomeStuff() is quite clumsy at the moment. It relies on a string parameter to decide what type of logger to create. A better design would be for doSomeStuff() to receive a factory object as a parameter, and then use the factory object to create the appropriate type of logger. To implement this factory mechanism, follow these steps:

* Define a new interface named LoggerProvider. The interface should have a single method named getLogger(), which returns a Logger object.
* [*aside*]---------------------------------------------------------------------------------------------------

In Java 7, the next step would be to define different implementations of LoggerProvider to create and return different types of logger. For example:

* + - You might define a class named BriefLoggerProvider, and implement getLogger() to create and return a BriefLogger instance.
    - You might define a class named VerboseLoggerProvider, and implement getLogger() to create and return a VerboseLogger instance.

In Java 8 onwards, you can avoid having to define specific factory classes. Instead, you can use constructor references to refer to the existing constructors in the BriefLogger and VerboseLogger classes. These constructors effectively fulfil the role of the getLogger() method, to create and return a BriefLogger or VerboseLogger object respectively. The following page describes how to make use of constructor references.

[*end aside*]----------------------------------------------------------------------------------------------

* Here's how to make use of constructor references:
  + In Main.java, change the signature of doSomeStuff() so that it receives a LoggerProvider object rather than a String.
  + Modify all the calls to doSomeStuff() so that instead of passing in a string, you pass in an appropriate constructor reference (i.e. VerboseLogger::new or BriefLogger::new). The compiler will treat the constructors as if they were implementations of the getLogger() method, to create and return the appropriate type of logger object.
  + Modify the code inside doSomeStuff() so that it uses the LoggerProvider to create the appropriate type of logger object.

When you've made all these changes, run the program again to ensure it works as before. There are several benefits to the code as it now stands:

* You've removed the messy and error-prone string tests in doSomeStuff().
* The code is more extensible. If you add new types of logger in the future, you don't need to worry about adding another branch to the string tests.
* You've avoided the overhead of traditional factory mechanisms in Java, which rely on a separate factory class for each kind of object that needs to be created (i.e. you didn't have to define classes such as BriefLoggerProvider and VerboseLoggerProvider). This can be a huge advantage in large-scale applications.

## Exercise 4 (If Time Permits): Defining static methods in an interface

In this exercise you'll implement the Null Object pattern in the logger application. For more information about this pattern, see:

* <http://en.wikipedia.org/wiki/Null_Object_pattern>

To understand the raison d'être of the Null Object pattern, follow these steps:

* In main(), add another call to doSomeStuff(), passing in null for the LoggerProvider parameter (e.g. to indicate you don’t want any logging this time).
* Run the program. When the program reaches the point where you pass null into doSomeStuff(), it crashes when it tries to call getLogger() on the null reference.

To implement the Null Object pattern in the program, you can take advantage of Java's support for static methods in interfaces. Follow these steps:

* In the LoggerProvider interface, add a static method named getNullLogger(). Implement the interface so that it returns a Logger object with a do-nothing implementation of the log() method (you can use a lambda expression to implement the do-nothing version of the log() method, if you like).
* Modify doSomeStuff() so that it checks if the incoming LoggerProvider parameter is null, and either creates a real logger or a "Null Object" logger as appropriate. Whichever type of logger is created, all the subsequent calls to log() are safe; they will either display brief messages, verbose messages, or no messages at all (in the case where a "Null Object" logger is being used).

Run the program. Verify the program displays (or doesn't display) log messages correctly.