Programming II

Object Oriented Programming with Java

- Advanced Topics -

Java 8: Functional Interfaces, Method References and Lambda Expressions

Alastair Donaldson

www.doc.ic.ac.uk/~afd

Remember good old `map` from Haskell?

```
map :: (a -> b) -> [a] -> [b]
```

Multiply each list element by 10

```
Prelude> map (\x -> 10*x) [0..9]
[0,10,20,30,40,50,60,70,80,90]
```

Convert each list element to a float

```
Prelude> map (\x -> 1.0*x) [0..9]
[0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
```

Raise list element x to a three-element list [x-1, x, x + 1]

```
Prelude> map (\x -> [x-1, x, x+1]) [0..9]
[[-1,0,1],[0,1,2],[1,2,3],[2,3,4],[3,4,5],[4,5,6],[5,6,7],
[6,7,8],[7,8,9],[8,9,10]]
```

Let's try to do this in Java

Attempt 1: three separate loops

```
public class JustLoops {

  public static void main(String[] args) {

    List<Integer> integers = new ArrayList<Integer>();
    for(int i = 0; i < 10; i++) {
      integers.add(i);
    }

    List<Integer> tenTimesBigger = new ArrayList<Integer>();
    for(Integer i : integers) {
      tenTimesBigger.add(10*i);
    }
    System.out.println(tenTimesBigger);

    List<Float> floats = new ArrayList<Float>();
    for(Integer i : integers) {
      floats.add(new Float(i));
    }
    System.out.println(floats);
```

Make the list [0..9]

Multiply elements by ten

Turn elements into floats

Attempt 1: three separate loops

```
    ...

    List<List<Integer>> triples = new ArrayList<>();
    for(Integer i : integers) {
      triples.add(Arrays.asList(new Integer[] { i-1, i, i+1 }));
    }
    System.out.println(triples);
  }

}
```

Raise each element to a triple

Program output:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
[[-1, 0, 1], [0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6],
[5, 6, 7], [6, 7, 8], [7, 8, 9], [8, 9, 10]]
```

Critique of Attempt 1

Produces the same effect as the Haskell program

…but the `map` logic is re-implemented each time

Not satisfactory

Attempt 2: a Transformer interface

Haskell's `map` takes a function that transforms something of type `a` into something of type `b`:

```
map :: (a -> b) -> [a] -> [b]
```

Let's write `map` in Java, in terms of an **interface** that can transform something of type `a` into something of type `b`

…but we're in Java mode, so we'll use `S` and `T`, not `a` and `b`

Transformer interface

The interface is **generic** with respect to types `S` and `T`

```
public interface Transformer<S, T> {

    public T transform(S x);

}
```

`transform` says: “give me an `S` and I will give you back a `T`

Let us write our three transformers

Three Transformer implementations

```
public class TimesTenTransformer
    implements Transformer<Integer, Integer> {
  @Override
  public Integer transform(Integer x) {
    return x*10;
  }
}
```

```
public class IntegerToFloatTransformer
    implements Transformer<Integer, Float> {
  @Override
  public Float transform(Integer x) {
    return new Float(x);
  }
}
```

```
public class IntegerToTripleTransformer
    implements Transformer<Integer, List<Integer>> {
  @Override
  public List<Integer> transform(Integer x) {
    return Arrays.asList(new Integer[] { x - 1, x, x + 1 });
  }
}
```

Notice here that we provide concrete types for S and T when we implement **Transformer**

Implementing `map` using the `Transformer` interface

```
public interface Transformer<S, T> {


  public T transform(S x);


  public static <A, B>
  List<B> map(Transformer<A, B> transformer, List<A> input) {
    List<B> result = new ArrayList<>();
    for(A a : input) {
      result.add(transformer.transform(a));
    }
    return result;
  }

}
```

In Java 8, an **interface** can have **static methods**

This is good, because sometimes an interface is the most logical home for such a method

Our `map` method is not specific to any particular transformer, so it makes sense for it to live in the interface

Using map with our transformers

```
public class UsingInterfaces {

  public static void main(String[] args) {

    List<Integer> integers = new ArrayList<Integer>();
    for(int i = 0; i < 10; i++) {
      integers.add(i);
    }

    List<Integer> tenTimesBigger = Transformer.map(
                        new TimesTenTransformer(), integers);
    System.out.println(tenTimesBigger);

    List<Float> floats = Transformer.map(
                        new IntegerToFloatTransformer(), integers);
    System.out.println(floats);

    List<List<Integer>> triples = Transformer.map(
                        new IntegerToTripleTransformer(), integers);
    System.out.println(triples);
  }

}
```

Critique of approach 2

We succeeded in capturing the essence of `map`

Generics played a nice role in making this work

Painful: writing a separate class each time we need a transformer.

Especially if we need a simple transformer and we only need it once

Approach 3: anonymous classes

Instead of declaring `TimesTenTransformer` as a class:

```
public class TimesTenTransformer
    implements Transformer<Integer, Integer> {
  @Override
  public Integer transform(Integer x) {
    return x*10;
  }
}
```

…and then using this class:

```
List<Integer> tenTimesBigger = Transformer.map(
                        new TimesTenTransformer(), integers);
```

…we can declare the class at the point we wish to use it.

We don't even need to give the class a name: it can be **anonymous**

An anonymous version of TenTimesTransformer

```
List<Integer> tenTimesBigger = Transformer.map(
                      new Transformer<Integer, Integer>() {
                        @Override
                        public Integer transform(Integer x) {
                          return x*10;
                        }
                      }
                      , integers);
```

This declares a **nameless** class that implements the `Transformer` interface, and creates a single instance of this class

An anonymous version of IntegerToFloatTransformer

```
List<Float> floats = Transformer.map(
                     new Transformer<Integer, Float>() {
                       @Override
                       public Float transform(Integer x) {
                         return new Float(x);
                       }
                     }
                     , integers);
```

Says:

"Make an instance of a class that implements the `Transformer<Integer, Float>` interface, by defining the `transform` method as follows…"

An anonymous version of IntegerToTripleTransformer

Similar:

```
List<List<Integer>> triples = Transformer.map(
                new Transformer<Integer, List<Integer>>() {
                  @Override
                  public List<Integer> transform(Integer x) {
                    return Arrays.asList(new Integer[]
                      { x - 1, x, x + 1 });
                  }
                }, integers);
```

Critique of approach 3

Arguably less painful than approach 2: we did not have to write a separate class file for each transformer.

Instead we described the transformers “on demand” using anonymous classes.

But the anonymous class syntax is pretty verbose!

Functional interfaces

An interface is a **functional interface** if it declares exactly one abstract method.

To implement a functional interface, a class just needs to provide the single abstract method.

`Transformer` is a functional interface:

- One abstract method: `transform`
- Also one non-abstract, static method, `map`

Functional interfaces and method references

Suppose a method expects a parameter of type `I`, where `I` is a **functional interface**.

In Java 8:

- Instead of passing an instance of a class that implements `I`…
- …you can pass a **method** that matches the signature of the single abstract method associated with the functional interface

Approach 4: use method references

`map` expects a `Transformer` as its first argument

`Transformer` is a functional interface

Instead of passing a `Transformer<S, T>` to `map`, we can pass any method that:

- accepts a single argument of type `S`
- returns something of type `T`

Passing timesTen as a method reference

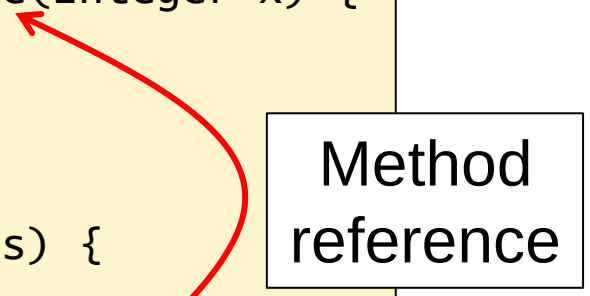
```
public class UsingMethodReferences {

  private static Integer timesTen(Integer x) {
    return x * 10;
  }

  public static void main(String[] args) {

    List<Integer> integers = ...;

    List<Integer> tenTimesBigger = Transformer.map(
      UsingMethodReferences::timesTen, integers);
    System.out.println(tenTimesBigger);

  }

}
```

Write `UsingMethodReferences::timesTen` to refer to static method `timesTen` of `UsingMethodReferences` class

There is also syntax for getting a reference to an instance method of an object (look it up)

Passing toTriple as a method reference

```
public class UsingMethodReferences {

  private static List<Integer> toTriple(Integer x) {
    return Arrays.asList(new Integer[]
      { x - 1, x, x + 1 });
  }

  public static void main(String[] args) {

    List<Integer> integers = ...;

    List<List<Integer>> triples = Transformer.map(
      UsingMethodReferences::toTriple, integers);
    System.out.println(triples);

  }

}
```



Passing Float constructor as a method reference

```
public class UsingMethodReferences {

  public static void main(String[] args) {

    List<Integer> integers = ...;

    List<Float> floats = Transformer.map(
                         Float::new, integers);
            System.out.println(floats);

  }

}
```

`Float::new` is a reference to the constructor `Float(Integer x)`

Type inference is used to determine which constructor makes sense here

Critique of approach 4

Passing in `Float::new` as a method reference is pretty neat!

It's not clear that passing in `timesTen` and `toTriple` as method references is worth it.

They do such simple things; it's annoying to have to write a specially named method for each of them.

Functional interfaces and lambda expressions

Suppose a method expects a parameter of type `I`, where `I` is a functional interface.

In Java 8, instead of passing an instance of a class that implements `I`, or a method reference, you can pass in a **lambda expression**.

A lambda expression describes a function that produces a result from some arguments.

Lambda expression: example

Instead of passing `map` a reference to the timesTen method:

```
List<Integer> tenTimesBigger = Transformer.map(
      UsingMethodReferences::timesTen, integers);
```

we can instead pass a lambda expression:

```
List<Integer> tenTimesBigger =
  Transformer.map(x -> x*10, integers);
```

Means we do not have to define `timesTen` at all!

Compactly describes a function which, given argument x, returns x*10

The term “lambda” comes from the Lambda Calculus

Approach 5: use lambdas

```
public class UsingLambdas {

  public static void main(String[] args) {

    List<Integer> integers = ...

    List<Integer> tenTimesBigger = Transformer.map(
                x -> x*10, integers);
    System.out.println(tenTimesBigger);

    List<Float> floats = Transformer.map(
                Float::new, integers);
    System.out.println(floats);

    List<List<Integer>> triples = Transformer.map(
                x -> Arrays.asList(new Integer[]
                     { x - 1, x, x + 1 })
                , integers);
    System.out.println(triples);
  }

}
```

These are lambda expressions

Critique of approach 5

Basically gives us the elegance of map in Haskell, but in Java.

Let's write a method to compose two transformers

Input: a transformer from `S` to `T`, a transformer from `T` to `U`

Output: the transformer from `S` to `U` obtained by composing the input transformers

```
public static <S, T, U>
Transformer<S, U> compose(
    Transformer<S, T> t1, Transformer<T, U> t2) {

    return (x -> t2.transform(t1.transform(x)));

}
```

Here we are returning a lambda expression. The caller of `compose` can then apply the lambda expression in the future

Composition in action

```
public class Composition {

  public static <S, T, U> Transformer<S, U> compose(
         Transformer<S, T> t1, Transformer<T, U> t2) {
    return (x -> t2.transform(t1.transform(x)));
  }

  public static void main(String[] args) {
    List<Integer> integers = ...;

    Transformer<Integer, List<Integer>> timesTenAndTriple =
      compose((x -> x*10),
              (x -> Arrays.asList(new Integer[]
                          { x - 1, x, x + 1 }))
            );
    List<List<Integer>> bigTriples =
                    Transformer.map(timesTenAndTriple, integers);
            System.out.println(bigTriples);
  }
}
```

Stores a reference to a lambda expression

The stored lambda expression is applied

Composition in action

Output:

```
[[-1, 0, 1], [9, 10, 11], [19, 20, 21], [29, 30, 31], [39, 40,
41], [49, 50, 51], [59, 60, 61], [69, 70, 71], [79, 80, 81],
[89, 90, 91]]
```

Summary

Anonymous classes avoid the need to declare a named class when only a simple activity is required.

A **functional interface** can be implemented in a more light-weight manner: by passing a method reference for the single abstract method.

A **lambda expression** can be used in place of a method reference, leading to very concise code

Take home message: none of this lets you do anything you could not already do.

It simply allows more elegant and concise code

Next time

A few more details about anonymous classes

A discussion of default methods (a.k.a. defender methods) in Java 8.