Java 8: Welcome to the 21st Century!

Bruce Cota

March 6, 2015

Java 8: Welcome to the 21st Century!

Why the snarky title?

- Lambdas are Java's version of closures.
- Closures have been around for at least 30 years (Perl, Haskell, Scheme, Lisp, ...)
- Closures have been popular in Javascript, Groovy, Scala, etc for 10 years or more.

What are Closures (Lambdas) for?

Closures are used to pass a behavior as a parameter.

Lambdas are mainly just syntactic sugar for something you can do with anonymous classes.

Violations of DRY

```
public String doGet(Institution institution, String path) throws ItemProcessingExcep
    Response response;
    String url = getUrl(institution, path);
    WebTarget resourceTarget = client.target(url);
    try {
        Invocation.Builder builder = getRequestBuilder(institution, resourceTarget);
        response = builder.get();
        validateResponse(response);
    } catch (Exception e) {
        throw new ItemProcessingException(e.getMessage());
    }
    String jsonString = response.readEntity(String.class);
    response.close();
    return jsonString;
}

public String doPost(Institution institution, String path, Item item) throws ItemPro
    Response response;
    String url = getUrl(institution, path);
    WebTarget resourceTarget = client.target(url);
    try {
        Invocation.Builder builder = getRequestBuilder(institution, resourceTarget);
        response = builder.post(Entity.entity(item, MediaType.APPLICATION\_JSON));
        validateResponse(response);
    } catch (Exception e) {
        throw new ItemProcessingException(e.getMessage());
    }
    String itemLocation = response.getHeaderString("Location");
    response.close();
    return itemLocation;
}
```

Violation of DRY

Two methods that really differ in only two lines:

```
        response = builder.get();
        ...
    String jsonString = response.readEntity(String.class);
```

vs

```
        response = builder.post(Entity.entity(item, MediaType.APPLICATION\_JSON));
        ...
    String itemLocation = response.getHeaderString("Location");
```

There are also PUT and DELETE methods.

DRY

To avoid the cut and paste, we need a method (say `callUrl`) that takes two additional pieces of information:

- How to generate a response (get or post?)
- What to return to caller (an entity or a the value of a Location header?)

Approach 1 (old school): Switch statements

```
String callUrl(Institution institution, String path, HttpMethod: callType) throws It
  ...
        switch (callType) {
          case GET:
             response = builder.get();
             break;
          case POST:
             response = builder.post(Entity.entity(item, MediaType.APPLICATION\_JSON
             break;
          ...
       }
  ..
  switch (callType) {
     case GET: return response.getHeaderString("Location");
     case POST: return response.getHeaderString("Location");
  }
}
```

Switch statements drawbacks

Switch statement could actually work in this case because

1. There are only four possible http methods
2. The return type is the same ("String") for all four Http Types.

But it has limited use. What if we can not predict all the possible values for the switch paramter?

Java 7 Solution

Create a new Interface to pass to `callUrl`. Let's do it functionally.

```
//A function from A to B
public interface Function<A,B> {
  A apply(B argument);
}

String callUrl(Institution institution, String path,
       Function<Invocation.Builder,Response> responseBuilder,
       Function<Response,String> responseProcessor)
   throws ItemProcessingException {
   Response response;
   String url = getUrl(institution, path);
    WebTarget resourceTarget = client.target(url);
      try {
          Invocation.Builder builder = getRequestBuilder(institution, resourceTarget);
          //APPLY FUNCTION ARGUMENT
          response = responseBuilder.apply(builder);
          validateResponse(response);
      } catch (Exception e) {
          throw new ItemProcessingException(e.getMessage(),e);
      }
      //APPLY FUNCTION ARGUMENT
      String responseString = responseProcessor.apply(response);
      response.close();
      return responseString;
```

Passing Behavior with Anonymous Classes

```
public String doGet(Institution institution, String path) throws ItemProcessingExcep

    Function<Invocation.Builder,Response> responseBuilder =
      new Function<Invocation.Builder,Response>() {
        public Response apply(Invocation.Builder builder) = {
          return builder.get();
        }
      }

    Function<Invocation.Builder,Response> responseProcessor =
      new Function<Response,String>() {
        public Response apply(Response response) = {
          return response.readEntity(String.class);
        }
      }


    return callUrl(institution, path, responseBuilder, responseProcessor);
}
```

Solution with Java 1.8 Lambda Syntax

```
//Our function interface is part of the standard Java library
import java.util.function.Function;

//callUrl is defined exactly as before

public String doGet(Institution institution, String path) throws ItemProcessingException
    Function<Invocation.Builder,Response> responseBuilder =
        (builder) -> builder.get();
    Function<Response,String> responseProcessor =
        (response) -> response.readEntity(String.class);
    return callUrl(institution, path, responseBuilder, responseProcessor);
}
```

Lambda Syntax

```
(builder) -> builder.get();
```

The above is syntactic sugar for this

```
new Function<Invocation.Builder,Response>() {
    public Response apply(Invocation.Builder builder)
        return builder.get();
    }
}
```

Lambda Syntax

```
(builder) -> builder.get();
```

The above is syntactic sugar for this

```
new Function<Invocation.Builder,Response>() {
    public Response apply(Invocation.Builder builder)
        return builder.get();
    }
}
```

Type Inference

```
Function<Invocation.Builder,Response> f
    = (builder) -> builder.get();
```

is equivalent to

```
Function<Invocation.Builder,Response> responseBuilder
    = (Invocation.Builder builder) -> builder.get();
```

The java compiler can *infer* the type of `builder` from the type of responseBuilder. In java 8, it's possible to leave out type declaration in some limited cases

Other Functional Interfaces

Lambda expressions denote implementations of *functional interfaces*, which you can write yourself with the `@FunctionalInterface` annotation. But there is a long list of predefined functional interfaces in `java.util.function`

For example, consider `Predicate<t>`

Predicate example

```
import java.util.ArrayList;
import java.util.function.Predicate;

class Filter {

    //This is a lousy filter, it's just for demonstrating predicate
    public <T> ArrayList<T> filter(ArrayList<T> items, Predicate<T> criteria) {
        ArrayList<T> newItems = new ArrayList<>();
        for (T item: items) {
            if (criteria.test(item)) {
                newItems.add(item);
            }
        }
        return newItems;
    }
}
```

Lambdas as Closures

Like inner classes, lambdas can refer to the environment in which they are defined, but with the same limitations. This compiles:

```
import java.util.function.Function;

public class Foo {
    public static void main(String[] args) {
        int x = 5;

        Function<Integer,Integer> f = (y) -> {
            return x+y;
        };

        System.out.println(f.apply(3));
    }
}
```

Composable Interfaces

```
import java.util.function.Function;

public class Foo {
    public static void main(String[] args) {
        Function<Integer,String> f = (x) -> { return Integer.toString(x); };
        Function<String,Integer> g = (y) -> { return Integer.parseInt(y); };
        Function <Integer,Integer> h = (z) -> { return g.compose(f).apply(z); };

        System.out.println(h.apply(3));
    }
}
```

Lambdas as Closures

Like inner classes, lambdas can refer to the environment in which they are defined, but with the same limitations. This does not compile:

```
import java.util.function.Function;

public class Foo {
    public static void main(String[] args) {
        int x = 5;

        Function<Integer,Integer> f = (y) -> {
            x = x+1;    //CANNOT DO THIS
            return x+y;
        };

        System.out.println(f.apply(3));
    }
}
```

This is the difference between closures and lambdas.

Summary

- Functional Interfaces and Lambdas provide a succinct way of factoring out common behaviors.
- It was possible to do this in JDK 7 with Anonymous classes, but they were too verbose.
- It may take practice to notice areas where use of lambdas is appropriate.

Addendum

- Java 8 has borrowed heavily from Scala.
- With `Streams` and use of functional interfaces, it might be possible to actually do functional programming in JDK 8.
- Immutability is probability out of the question, though :)
- The `Optional` type should be examined – can we use the compiler to prevent NPEs?

Does functional programming in Java warrant a COP or thread in existing COP?