Working with Lambdas

1. Working with Lambdas

Cyclops provides a number of classes that makes working with Java 8 lambda expressions a little easier, they including in the Lambda Helper Classes sidebar.

Lambda Helper Classes

1. Mutable : a wrapper class for a mutable local (or external value)
2. LazyImmutable : a value that can be set just once, and computed lazily
3. ExceptionSoftener : because JDK 8 functional interfaces don't support checked exceptions, ExceptionSoftener allows CheckedExceptions to be Softened to Unchecked Exceptions without impacting their type (equiavlent to Lombok's Sneaky Throws) - and provides a number of static methods for Softening the jOOλ Checked Functional Interfaces.
4. Memoize : A class that allows functional interfaces & method references to cache their return values, supporting pluggable cache implementations (such as Caffeine and Guava)
5. Curry : Convert a single mutli-parameter function (or method reference) into a chain of single parameter functions
6. Uncurry : Convert a chain of single-parameter functions into a single multi-parameter function
7. Lambda : A class with static to help with type inferencing when using Lambda's
8. PartialApplicator : Lazily set some of the parameters on a multi-parameter function, returning a function with the remaining parameters.

1.1. Mutable

Overview

Java lambda expressions can access local variables, but the Java compiler will enforce an **effectively final** rule. cyclops-closures makes capturing variables in a mutable form a little simpler. Mutable[1] provides a wrapper over a mutable variable, it implements Convertable[2] which allows the value to be converted into various forms (such as a thread-safe AtomicReference, Optional, Stream, CompletableFuture etc).

Available Mutable classes

- Mutable[3]
- MutableInt[4]
- MutableDouble[5]
- MutableLong[6]
- MutableFloat[7]
- MutableShort[8]
- MutableByte[9]
- MutableChar[10]
- MutableBoolean[11]

Mutable local variables

Mutable can be used to work around Java's effectively final rule, simply wrap any Mutable variable you would like to mutate inside an (effectively) final Mutable instance.

[1] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/Mutable.html
[2] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/Convertable.html
[3] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/Mutable.html
[4] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableInt.html
[5] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableDouble.html
[6] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableLong.html
[7] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableFloat.html
[8] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableShort.html
[9] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableByte.html
[10] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableChar.html
[11] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/mutable/MutableBoolean.html

Example 1. MutableInt within a Stream

In this example, we mutate a local primitive variable using MutableInt, inside a lambda expression passed into a Stream.

```
MutableInt num = MutableInt.of(20);

Stream.of(1,2,3,4)
      .map(i->i*10)
      .peek(i-> num.mutate(n->n+i))
      .forEach(System.out::println);

assertThat(num.get(),is(120));
```

The Mutable classes are not suitable for multi-threaded use, for example within parallel Streams, however they do implement the Converable interface which allows values to be easily converted into many different types including AtomicReference.

Example 2. Set inside a lambda

In this simple example we will create a Mutable that manipulates Objects - in this case with the generic type parameter of <String>, and we will set the value of the mutable inside a Runable.

```
Mutable<String> var =  Mutable.of("hello");
Runnable r = () -> var.set("world");
```

In the above example, the value stored inside of var will not be set until the run method on r is called.

Mutable external variables

Mutable can also be used to mutate non-local variables such as fields, or even fields in other objects.

Example 3. Create a Mutable from a Supplier and Consumer combination

Mutables can be used to wrap access to an external field/s via the fromExternal method with a Supplier and Consumer.

In the example below, the call to ext.set( ) updates the field var - via the consumer passed as the second parameter to fromExternal.

```
String var = "world";

Mutable<String> ext = Mutable.fromExternal(()->var,v->this.var=v);
ext.set("hello");
```

In addition we can apply functions to transofrm both our inputs and outputs. For example if we want to create different mutable instances to handle setting the same source, in different ways.

```
String var = "world";

Mutable<String> ext = Mutable.fromExternal(()->var,v->this.var=v);
ext.set("hello");

Mutable<String> userInputHandler = ext.mapInputs(in-> validate(in));
userInputHandler.set("hello"); // will be validated before setting var
```

To use an external Mutable to update a local value, that local var itself would have to be stored in a Mutable.

```
Mutable<String> var = Mutable.of("world");

Mutable<String> ext = Mutable.fromExternal(()->var.get(),v->this.var.set(v));
ext.set("hello");
```

Usages of mutable in Cyclops

Mutable is used inside Cyclops for-comprehensions simplify the handling of an immutable (persisent) datastructure that needs to be mutated.

Example 4. Mutable is used to store the current variables in a for-comprehension

```
build(ComprehensionData c, Function f) {

 Mutable<PVector<String>> vars = new Mutable<>(TreePVector.empty());
 getAssigned().stream().forEach(e->
 addToVar(e,vars,handleNext(e,c,vars.get())));
 Mutable<Object> var = new Mutable<>(f);

 return c.yield(()-> unwrapNestedFunction(c, f, vars.get());
}
```

1.2. LazyImmutable

A set-once wrapper over an AtomicReference. Unlike the MutableXXX classes LazyImmutable is designed for sharing across threads where the first thread to attempt can write to the reference, and subsequent threads can read only. LazyMutable[12] provides a thread-safe wrapper over a variable that can be set once, it implements Convertable[13] which allows the value to be converted into various forms (such as a thread-safe AtomicReference, Optional, Stream, CompletableFuture etc).

⚠ Only the first attempt at setting a value is accepted, subsequent attempts are ignored.

Usage

We use LazyImmutable inside of cyclops itself to implement Memoization (lambda caching) support. We do this by taking advantage of lazy evaluation support inside LazyImutable. The example below shows how it is used.

[12] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/immutable/LazyImmutable.html
[13] http://static.javadoc.io/com.aol.cyclops/cyclops-closures/7.1.0/com/aol/cyclops/closures/Convertable.html

Example 5. Create a memoizing (caching) Supplier that can be shared across threads.

Inside our memoizeSupplier method we use a local LazyImmutable to lazily cache the result of calling s.get();

```
public static <T> Supplier<T> memoizeSupplier(Supplier<T> s){
  LazyImmutable<T> lazy = LazyImmutable.def();
  return () -> lazy.computeIfAbsent(s);
}

Supplier<String> cached = memoizeSupplier(()->"Hello
 world:"+System.currentTimeMillis());
```

When cached.get() is called for the first time, it delegates to lazy.computeIfAbsent(s);. Our LazyImmutable will not be set at this point and it will execute and cache the result of s.get();

Subsequent calls to cached.get() will all show the same timestamp as the cached value will be used.

By using computeIfAbsent we can have LazyImmutable lazily determine whether or not the value to set should be computed.

Strict / non-lazy usage

The setOnce method provides a non-lazy (strict) alternative to computeIfAbsent. In this case the value to be passed is always evaluated, but the setOnce (simulated Immutability) semantics are maintained. In other words if setOnce is called multiple times with different values, the LazyImmutable will continue to hold only the first.

Example 6. A non-lazy LazyImmutable by using setOnce.

setOnce - sets a value directly, but only the first time it is called

```
LazyImmutable<Integer> value = new LazyImmutable<>();
Supplier s= () -> value.setOnce(10).get();

assertThat(s.get(),is(10));
assertThat(value.get(),is(10));
```

computeIfAbsent lazily compute a value if the lazyimmutable is unset

```
LazyImmutable<Integer> value = new LazyImmutable<>();
Supplier s= () -> value.computeIfAbsent(()->10);
assertThat(s.get(),is(10));
assertThat(value.computeIfAbsent(()->20),is(10));
```

set twice, second time has no effect

```
LazyImmutable<Integer> value = new LazyImmutable<>();
Supplier s= () -> value.setOnce(10);
value.setOnce(20); //first time set

s.get();


assertThat(value.get(),is(20));
```

Monad-like functionality

LazyImmutable also has monadic functional operators such as map & flatMap, these will be familar to Java developers who have experience using Optional or Stream. They can be used to create a new LazyImmutable with a transformed value inside.

Example 7. flatMapping a LazyImmutable.

```
//flatMap
LazyImmutable<Integer> value = new LazyImmutable<Integer>();
value.setOnce(10);
LazyImmutable<Integer> value2 = value.flatMap(i->LazyImmutable.of(i+10));
assertThat(value2.get(),equalTo(20));
```

1.3. ExceptionSoftener

The 'problem' with functional interfaces

JDK Functional interfaces do not support CheckedExceptions.

```
public Data load(Task t) throws IOException(){
    ..
}

Stream.generate(()->nextTask())
      .map(this::load)  // DOES NOT COMPILE
```

Overview

With Cyclops ExceptionSoftener, there is no need to declare CheckedExceptions, or even to wrap them inside RuntimeException. The ExceptionSoftener converts CheckedExceptions into UncheckedExceptions *without* changing the Exception type. That is, your function or method can still throw IOException, it just no longer needs to declare it.

The example below shows a number of usages of ExceptionSoftener.

Example 8. Throwing a softened exception

```
throw ExceptionSoftener.throwSoftenedException(new IOException("hello"));

throw ExceptionSoftener.throwSoftenedException(new Exception("hello"));

//doesn't need softened, but will still work
throw ExceptionSoftener.throwSoftenedException(new RuntimeException("hello"));
```

Always use **throw** ExceptionSoftener.throwSoftenedException, where you would throw an actual Exception directly, rather than just passing the exception directly into the softener. This lets the compiler know an Exception is being thrown at this point, and means you won't get compile time errors about missing return values at an unreachable point in the code.

The JDK functional interfaces don't support CheckedExceptions, so the ExceptionSoftener can prove very useful when working with those.

ExceptionSoftener provides softenXXX methods for all Checked Functional interfaces in jOOλ[14]

[14] http://www.jooq.org/products/jOO%CE%BB/javadoc/0.9.7/org/jooq/lambda/fi/util/function/package-frame.html

Example 9. soften an IOException

Example, softening an IOException. This method will continue to throw an IOException, but no longer needs to declare it.

```
public Data load(String input) {
        try{
          //do something
        }catch(IOException e) {
            throw ExceptionSoftener.throwSoftenedException(e);
        }
}
```

In the above example IOException can be thrown by load, but it doesn’t need to declare it.

Wrapping calls to methods

With functional interfaces and lambda’s

Where we have existing methods that throw softened Exceptions we can capture a standard Java 8 Functional Interface that makes the call and throws a a softened exception

Example 10. Soften a method that throws a CheckedException to a plain function

```
Function<String,Data> loader = ExceptionSoftener.softenFunction(file-
>load(file));

public Data load(String file) throws IOException{
     ///load data
}
```

Example 11. Soften inside a stream

```
Stream.of("file1","file2","file3")
      .map(ExceptionSoftener.softenFunction(file->load(file)))
      .forEach(this::save)
```

We can simplify further with method references.

```
Data loaded = ExceptionSoftener.softenFunction(this::load).apply(fileName);

Stream.of("file1","file2","file3")
      .map(ExceptionSoftener.softenFunction(this::load))
      .forEach(this::save)

public String load(String file) throws IOException{
        throw new IOException();
}
```

Example 12. Soften a Supplier

```
Supplier<String> supplier = ExceptionSoftener.softenSupplier(this::get);

assertThat(supplier.get(),equalTo("hello"));

private String get() throws IOException{
  return "hello";
}
```

ExceptionSoftener is used extensively within Cyclops and simple-react.

Example 13. Soften in a retry Function from Cyclops

This example comes from cycops-streams, by using SoftenRunnable we can use Thread.sleep without having to declare a throws / try & catch block for InteruptedException. Any exception caught from catching the users supplied function can also be thrown upwards.

```
Function<T,R> retry = t-> {
  int count = 7;
  int[] sleep ={2000};
  Throwable exception=null;
  while(count-->0){
   try{
    return fn.apply(t);
   }catch(Throwable e){
    exception = e;
    }
    ExceptionSoftener.softenRunnable(()->Thread.sleep(sleep[0]));

    sleep[0]=sleep[0]*2;
  }
  throw ExceptionSoftener.throwSoftenedException(exception);

};
```

1.4. Memoization

Memoisation allows us to transparently cache the result of function calls. With Cyclops[15] we can memoise any JDK 8 Function via Memoise.memoiseFunction[16] (and by extension--via method references, we can also memoise most Java methods too!). For example

Example 14. Memoize a simple addition function

```
int called =0; //instance variable
```

```
Function add = a->a + ++called;
```

We can memoize our add function as follows

```
Function memoized = Memoise.memoizeFunction(add);
```

Repeatedly calling memoised with a single value, will not result in called being incremented.

```
assertThat(memoized.apply(0),equalTo(1));
assertThat(memoized.apply(0),equalTo(1));
assertThat(memoized.apply(0),equalTo(1));
```

But, of course the memoisation is specific to the input parameter. Recalling memoised with a new value (say 1) will result in call being incremented, the first time we make that new call.

[15] https://github.com/aol/cyclops
[16] http://www.javadoc.io/doc/com.aol.cyclops/cyclops-functions/5.0.0

```
assertThat(s.apply(1),equalTo(3));
assertThat(s.apply(1),equalTo(3));
```

Memoizing method calls

com.aol.cyclops.functions.Memoize contains a number of methods for memoising JDK 8 Functional interfaces. Supplier, Callable, Function, BiFunction and Predicates. Cyclops Memoize class makes it simple to cache the result of method calls.

See also Memoisation,-Currying,-Uncurrying-and-Type-Inferencing[17]

Example 15. Memoize a method with four parameters

```
int called = 0; // instance variable

QuadFunction cached = Memoize.memoizeQuadFunction(this::addAll);

assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(called,equalTo(1));

private int addAll(int a,int b,int c, int d){
    called++;
    return a+b+c+d;
}
```

Cleaner type inference

Via Lombok val[18] (entirely optional)

Example 16. Scala-like type inference with Lombok

```
int called = 0; // instance variable

val cached = memoizeQuadFunction(this::addAll);
```

[17] https://github.com/aol/cyclops/wiki/Memoisation,-Currying,-Uncurrying-and-Type-Inferencing
[18] https://projectlombok.org/features/val.html

```
assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(cached.apply(1,2,3,4),equalTo(10));
assertThat(called,equalTo(1));


private int addAll(int a,int b,int c, int d){
 called++;
 return a+b+c+d;
}
```

Always check IDE Compatibility with any Lombok operators you use. Lombok is an annotation preprocessor, it doesn't introduce a runtime dependency for your project. However, while all annotations / keywords work with Eclipse - the same is not true for other IDEs. Delombok can remove Lombok annotations replacing them in your source with equivalent code.

Example 17. Memoize a supplier

Cyclops supports Memoization for a large range of Java Functional Interfaces, in this example we memoize a supplier.

```
Supplier<Integer> s = memoiseSupplier(()->++called);
assertThat(s.get(),equalTo(1));
assertThat(s.get(),equalTo(1));
```

Memoization in Microserver

Microserver[19] uses Cyclops memoization to ensure that plugins are only ever loaded once.

Example 18. Ensure plugins are loaded once in Microserver

```
public class PluginLoader {

 public final static PluginLoader INSTANCE = new PluginLoader();
```

19 https://github.com/aol/micro-server

```
 public final Supplier<List<Plugin>> plugins =
                                    Memoize.memoizeSupplier(this::load);

 private List<Plugin> load(){
   return  SequenceM.fromIterable(ServiceLoader.load(Plugin.class)).toList();
 }
}
```

Referential Transparency & Cyclops Memoization

Referential Transparency is an academic term that means that for any given input a function will always return the same output - in any context, and will not affect state outside of the function. In other words a call to the function can be replaced with the value it returns.

Cyclops offers two forms of Memoization, one of which is suitable for referentially transparent (or pure) functions, and the other which may be appropriate with impure functions (those for which a given input may not always map to the same output).

To support that later, impure type of function, cyclops supports Memoization with pluggable caches. Java is not a functionally pure language and we feel supporting this type of caching is useful for Java developers.

Memoization with plugabble caches

By default a Memoized lambda or method reference will cache the return value inside the instance until it is cleared by the garbage collector.

simple-react[20] supports auto-memoization of functions within a Stream, and this is implemented via cyclops-memoization.

Example 19. Configure auto-memoization in simple-react with a ConurrentHashMap

```
Map cache = new ConcurrentHashMap<>();
LazyReact react = new LazyReact().autoMemoizeOn((key,fn)->
 cache.computeIfAbsent(key,fn));
```

[20] https://github.com/aol/simple-react

```
List result = react.of("data1","data1","data2","data2")
                .map(i->calc(i))
                .toList();
```

It is also possible to use advanced modern caching libraries such as Caffeine or Guava.

Example 20. Configure auto-memoization in simple-react with a Guava cache

```
//configure LRU cache with max time to live
Cache<Object, String> cache = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .build();

LazyReact react = new LazyReact().autoMemoizeOn((key,fn)-> cache.get(key,()-
>fn.apply(key));
List result = react.of("data1","data1","data2","data2")
                .map(i->calc(i))
                .toList();
```

1.5. Currying & Uncurrying

Currying

Currying involves creating a ‘chain’ of functions, were arguments are evaluated one-by-one, where each apply call results in either another single argument function or the final result. This contrasts with partial application (above) which may produce a single function that accepts multiple parameters. Curried functions always only accept one parameter at a time.

Example 21. Currying a String concatanation function

Given a method or function that performs String concatanation over 3 Strings

```
TriFunction<String, String, String, String> concat = (a, b, c) ->
                                                        a + b + c;
```

or

```
TriFunction<String, String, String, String> concat = this::concatMethod;

public String concatMethod(String a, String b, String c){
    return a+b+c;
}
```

Using Curried Functions our String concatanation example would like

```
Function<String,Function<String,Function<String,String>>> curried =
 Curry.curry3( concat);
```

Which is very verbose. We can simplify this using Lombok's type inferencing val keyword

```
val curried =  Curry.curry3( concat);
```

In practice, if you are not making use of Lombok, it is cleaner to using Currying in a point free style, that is to Curry a function and pass it is a parameter to another function (that can defined the function chain in a cleaner way with Generics).

point-free style Is a programming style where the program flows in a fluent style from one function call to the next without individually defining return values or arguments.

In addition at the point of currying one or more parameters may be applied.

Example 22. Partially applying parameters

```
Function<String,Function<String,String>> oneApplied =
 Curry.curry3( concat).apply("hello");

Function<String,String> twoApplied =
 Curry.curry3( concat).apply("hello").apply("world");
```

The syntax for a Curried function looks something like this

```
(String a) -> (String b) -> (String c) -> b + a + c;
```

Or without types

```
a -> b -> c -> b + a + c;
```

Where the arrow syntax is simply the lambda expression arrow. Here we are defining a lambda, that accepts an Integer and returns another lambda (that in turn accepts and returns a String).

The Cyclops Lambda class can help with creating curried functions (although types still have to be specified).

Using Currying to show nesting

Another place in Cyclops where Currying shows up, is inside For Comprehensions--where the Curried syntax is chosen specifically to show nesting levels. E.g.

Example 23. Currying to show nesting

In this example we can show the levels of nesting via currying

```
person -> car -> insurance -> { }
```

Currying can be very useful in conjunction with Cyclops for-comprehensions and existing methods, use the appropriate Curry method to create a curried reference to fit the yield or filter opertors!

Cyclops can convert any function (with up to 8 inputs) or method reference into a chain of one method functions (Currying). This technique is a useful (and more safe) alternative to Closures. The Curried function can be created and values explicitly passed in rather than captured by the compiler (where-upon they may change).

Example 24. Currying method references

```
import static com.aol.cyclops.functions.Curry.*;

curry2(this::mult).apply(3).apply(2);
//6

public Integer mult(Integer a,Integer b){
 return a*b;
}
```

Example 25. Currying a BiFunction

```
Curry.curry2((Integer i, Integer j) -> "" + (i+j)
 + "hello").apply(1).apply(2);

//"3hello"
```

Curry Consumer

The CurryConsumer class allows Consumers to also be Curried.

Example 26. Currying a consumer

```
CurryConsumer.curry4( (Integer a, Integer b, Integer c,Integer d) -> value = a
+b+c+d).apply(2).apply(1).apply(2).accept(3);

//8
```

1.6. Uncurrying

Uncurrying is the process of converting a chain of single-parameter functions into a single multi-parameter function (i.e. it is the reverse of Currying).

com.aol.cyclops.functions.Uncurry has methods to uncurry nested curried Functions of up to 8 levels deep. com.aol.cyclops.functions.UncurryConsumer does the same thing for curried Consumers up to 5 levels deep.

Example 27. Uncurrying in place example

```
Uncurry.uncurry3((Integer a)->(Integer b)->(Integer c)->a+b+c).apply(1,2,3)
//6
```

Example 28. Example Uncurrying a function to a function that takes 4 parameters

```
Uncurry.uncurry4((Integer a)->(Integer b)->(Integer c)->(Integer d)->a+b+c+d)
    .apply(1,2,3,4)
//10
```

Uncurry Consumer

com.aol.cyclops.functions.CurryConsumer provides methods to curry Consumers of up to 8 parameters.

Example 29. Example Uncurrying a consumer to a consumer that takes 4 parameters

```
UncurryConsumer.uncurry2((Integer a)->(Integer b) -> value = a
+b ).accept(2,3);
assertThat(value,equalTo(5));
```

1.7. Partial Application

We can also create partially applied functions. These are functions were the some of the input values to a function are provided up front, but not all. It converts, for example, a function that takes 3 input parameters, into a function that takes only 1. E.g. .partially applying values to a String concatonation function

Given the following function that concatonates three Strings

```
TriFunction<String, String, String, String> concat = (a, b, c) ->
                                                       a + b + c;
```

We can create a partially applied concatanator that will concat a supplied parameter to “hello” and “world” e.g.

```
Function<String, String> pa = PartialApplicator.partial3(“Hello”
                                     ,“World”, concat);
```

Using our new concatonator function (pa) with “!!!” should give use “Hello World!!!”

```
assertThat(concatStrings.apply(“!!!”), equalTo(“Hello World!!!”));
```

1.8. Type inferencing

The class com.aol.cyclops.lambda.utils.Lambda provides static helper methods for defining curried Lambda expressions of up to 8 nested Functions.

This is useful for creating anonymous functions where Java’s type inferencing won’t normally be able to infer types & for use in conjunction with Lombok’s val keyword which infers types from the right hand side of an expression.

Example 30. Anonymous function example

```
import static com.aol.cyclops.functions.Lambda.*;

Mutable myInt = Mutable.of(0);

Lambda.l2((Integer i)-> (Integer j)-> myInt.set(i*j)).apply(10).apply(20);

//myInt.get() : 200
```

Example 31. Lombok val example

```
val fn  = l3((Integer a)-> (Integer b)->(Integer c) -> a+b+c)
```