

COMP 322: Parallel and Concurrent Programming

Lecture 3: Higher Order Functions

Mack Joyner
mjoyner@rice.edu

<http://comp322.rice.edu>



Since Java 8: Function interface

```
var oneplus = new Function<Integer, Integer>(){  
    public Integer apply(Integer x){  
        return x + 1;  
    }  
};
```

```
Function<Integer, Integer> oneplus2 = x -> x + 1;
```

```
System.out.println(oneplus.apply(5));  
System.out.println(oneplus2.apply(7));
```



Operating on a list

Last lecture worksheet: functional, recursive

```
List<Integer> originalList = . . .  
var resultList = evens(originalList);  
  
static GList<Integer> evens(GList<Integer> input){  
  if (input.isEmpty()) return input;  
  return input.head() % 2 == 0  
    ? evens(input.tail()).prepend(input.head())  
    : evens(input.tail());
```



Operating on a list

Even better:

```
static GList<Integer> evens(GList<Integer> input) {  
    return input.filter(i -> i % 2 == 0);  
}
```

lambda expression

another operation on GList (just like prepend)



Some FP terminology

First-class functions: functions can be treated as any other variable

Assigned to any other variable

Passed as an argument to a function

Returned as a result from a function

First order functions: “normal” functions that do not take other functions as arguments or return a function as a result

Higher-order functions: functions that take other functions as arguments or return a function as a result

Only possible if functions are first-class



repeatFunc

We'd like to “repeat” a function multiple times

```
Function<Integer,Integer> oneplus = x -> x + 1;  
assertEquals(2, oneplus.apply(1));
```

```
Function<Integer,Integer> fourplus = repeatFunc(oneplus, 4);  
assertEquals(5, fourplus.apply(1));
```

repeatFunc needs to be a higher-order function: it takes a function as an argument, returns another function



Implementing *repeatFunc*

```
Function<Integer,Integer> oneplus = x -> x + 1;
```

```
static <T> Function<T,T> repeatFunc(Function<T,T> f, int n) {  
    if(n == 0) {  
        return x -> x;  
    } else {  
        return x -> f.apply(repeatFunc(f, n-1).apply(x));  
    }  
}
```

```
var fourplus = repeatFunc(oneplus, 4);
```

```
Function<String,String> doublestring = x -> x.concat(x);
```

```
var fourconcat = repeatFunc(doublestring, 4);
```



Functions are objects with methods

We can directly compose functions without lambda syntax

Methods defined on *java.util.function.Function* help make clean code.

```
static <T> Function<T,T> repeatFunc(Function<T,T> f, int n) {  
    if (n == 0) {  
        return Function.identity();  
    } else {  
        return f.compose(repeatFunc(f, n-1));  
    }  
}
```

Java code now looks math-ish!

$$\text{identity}(x) = x$$
$$\text{repeat}(f, n) = \begin{cases} \text{identity}, & \text{if } n = 0 \\ f \circ \text{repeat}(f, n - 1), & \text{otherwise} \end{cases}$$



repeatFunc is a higher-order function

- “Higher order function”: a function on functions!
 - This includes *Function.compose / andThen*, as well as our *repeatFunc*
 - *a.compose(b)* \equiv
 - *b.andThen(a)* \equiv
 - $x \rightarrow a.apply(b.apply(x))$
- In Java, “everything is an object”
- Functions are just objects
 - with *apply* methods (the lambda body)
 - with composition methods
 - with cool lambda syntax to make them



Neat trick: Lexical scope

The lambda hangs onto f and n , after *repeatFunc* exits!

```
static <T> Function<T,T> repeatFunc(Function<T,T> f, int n) {  
    if(n == 0) {  
        return x -> x;  
    } else {  
        return x -> f.apply(repeatFunc(f, n-1).apply(x));  
    }  
}
```

Closure: a lambda expression that captures values from the outer lexical scope



Java's lambda lexical scope rules

- A lambda can “capture” any primitive type (int, double, etc.) variables
 - Makes a copy, keeps it internally.
- A lambda can “capture” any *final* object-typed variables
 - Error if the variable might mutate (“effectively final” is required).
 - It really makes a copy of what the variable points to.
 - Would have been super-confusing if you could change the “outside” variable and the “inside” variable didn’t change.



Using lambdas: the GList filter function

```
public GList<T> filter(Predicate<T> predicate) {  
    if (predicate.test(headVal)) {  
        return tailVal.filter(predicate).prepend(headVal);  
    } else {  
        return tailVal.filter(predicate);  
    }  
}
```

An instance of the Predicate interface (a *lambda* returning boolean)

Applying the function



Functional programming vocabulary

- Predicate: a function that returns a boolean
- Operator: a function returning the same type
 - unary operator: one argument (e.g., trig functions)
 - binary operator: two arguments (e.g., addition, subtraction)
- Function: arguments and results can be different types
- Supplier: *produces* data (e.g., reading lines of text from a file), no input
- Consumer: eats data, has side effect (e.g., printing), returns nothing



Various forms of lambda syntax

These are all equivalent:

```
public class Foo {  
  // "inline" lambdas  
  Function<Integer,Integer> oneplus1 = x -> x + 1;  
  Function<Integer,Integer> oneplus2 = (x) -> { return x + 1; };  
  Function<Integer,Integer> oneplus3 = (Integer x) -> (x + 1);
```

```
static int oneplusx(int x) {  
  return x + 1;  
}
```

Note: Integer (the “object type”) rather than int (the “primitive

```
// "method reference" lambda (works for static and instance methods)  
Function<Integer,Integer> oneplus4 = Foo::oneplusx;
```



Common FP list operators

list.map(function) → list

Apply the function to every element of the list, return a new list

list.filter(predicate) → list

Compute a new list: every element where the predicate is true

list.fold(zero, function(accumulator, element)) → value

Apply the function to all elements, accumulating the value along the way

Also known as ***reduce***

list.sort(comparator) → list

Return a new list sorted by a function that says which is bigger



Mapping

Replace each element in a list with the function applied to it

```
GList<Integer> originals = ... ;  
GList<Integer> squares = originals.map(i -> i * i);
```

```
GList<String> strings = ...;  
GList<String> lowercases = strings.map(x -> x.toLowerCase());  
GList<Integer> lengths = strings.map(String::length);
```

Function return type can be different (e.g., `String::length`)

Python's list comprehensions: equivalent to our *filter*, then *map*



map, implemented

```
class Cons<T> implements GList<T> {  
  
    public <R> GList<R> map(Function<T, R> f) {  
        return tailVal.map(f).prepend(f.apply(headVal));  
    }  
}
```

```
class Empty<T> implements GList<T> {  
  
    public <R> GList<R> map(Function<T, R> f) {  
        return empty();  
    }  
}
```



Filter

“Pick” which elements of the original list to keep

```
class Cons<T> implements GList<T> {
  public GList<T> filter(Predicate<T> predicate) {
    if (predicate.test(headVal)) {
      return tailVal.filter(predicate).prepend(headVal);
    } else {
      return tailVal.filter(predicate);
    }
  }
}
```

```
class Empty<T> implements GList<T> {
  public <R> GList<R> filter(Predicate<T> p) {
    return empty();
  }
}
```



Folding

Let's say we have a list of numbers to add: { 1, 2, 3, 4, 5, 6, 7, 8 }

What order should we add them?

Fold-left: $(((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8$

Fold-right: $1 + (2 + (3 + (4 + (5 + (6 + (7 + 8))))))$

For integer addition, we get the same answer, but not for others

```
GList<Integer> intList = ...
```

```
int sum = intList.foldRight(0, (x, y) -> x + y);
```

zero / default value for empty list



Commutativity & associativity

Commutativity: $\forall_{a,b} : a + b = b + a$

Associativity: $\forall_{a,b,c} : a + (b + c) = (a + b) + c$

Integer math: associative and commutative

FoldLeft and FoldRight will give you identical answers

String concatenation: associative, but not commutative

FoldLeft and FoldRight can give you identical answers, if you're careful

Otherwise, you'll end up reversing the order of the strings



Many uses for folding

List of strings

Join into one string, find longest string, find shortest string, etc.

List of integers

Minimum, maximum, average, sum, etc.

List of bitmap images

Overlay images, concatenate horizontally, etc.

Vocabulary note: “fold” and “reduce” are synonyms. If you’ve heard of “MapReduce”, this is broadly how it works. (More in the coming lectures.)



foldRight, implemented

Fold-right: $1 + (2 + (3 + (4 + (5 + (6 + (7 + 8))))))$

```
class Cons<T> implements GList<T> {  
    public <U> U foldRight(U zero, BiFunction<T, U, U> operator) {  
        return operator.apply(headVal, tailVal.foldRight(zero, operator));  
    }  
}
```

```
class Empty<T> implements GList<T> {  
    public <U> U foldRight(U zero, BiFunction<T, U, U> operator) {  
        return zero;  
    }  
}
```



Summary

Functions are first-class objects in Java (since Java 8)

Enables functional programming paradigm in Java

Just Java interfaces and objects, with convenient lambda syntax

Closures are lambdas that capture the values from the outer lexical scope

Only primitive variables or effectively final objects can be captured

Higher-order functions take other functions as arguments and/or return functions as result

Very powerful concept, allows us to create generic functions that can perform all kinds of things

Higher-order functions Hall of Fame: *map*, *filter*, *fold*

