Abstracting Control Structures to Control the Application Flow

Esteban Herrera
JAVA ARCHITECT

@eh3rrera www.eherrera.net

Imperative Control Structure

```
List<Product> products = order.getProducts();
Double discount = 0.0;
for (Product p : products) {
    if (this.isGiftProduct(p)) {
        discount = p.getPrice();
        break;
    }
}
```

Using High-order Functions

```
List<Product> products = order.getProducts();
Double discount = products.stream()
                    .filter(this::isGiftProduct)
                    .map(Product::getPrice)
                    .findFirst()
                    .orElse(0.0);
```

Using High-order Functions and Ternary Operator

```
List<Product> products = order.getProducts();
Double discount = products.stream()
                .mapToDouble(p -> this.isGiftProduct(p)
                              ? p.getPrice() * 0.5
                              : p.getPrice())
                .average()
                .orElse(0.0);
```

```java
if (isGiftProduct(p)) {
    return p.getPrice() * 0.5;
} else {
    return p.getPrice();
}
```

```java
Function<Product, Double> price = p -> {
    if (isGiftProduct(p)) {
        return p.getPrice() * 0.5;
    } else {
        return p.getPrice();
    }
}
```

The Optional Type

Value?

```
optionalElement
          .ifPresent(val -> doSomethingWithValue(val)),
          .orElseGet(() -> doSomethingElse());
```

You Have to Decide if It Is Worth the Effort

Does this make the code more readable?

Is this an over-engineered implementation?

Using Recursion Instead of Loops

Loops Iterate over Lists

This is a list from 1 to 10

```
for (int i = 1; i <= 10; i++) {
    // ...
}
```

Whatever you can do with recursion, you can also do it with iteration, and vice versa.

$$[\overbrace{1}^{\text{head}}\quad \overbrace{2\quad 3\quad 4}^{\text{tail}}]$$

$$1 + [\,2\quad 3\quad 4\,]$$

$$[\overbrace{1}^{\text{head}}\quad\overbrace{2\quad 3\quad 4}^{\text{tail}}]$$

$$1 + [\,2\quad 3\quad 4\,]$$

$$2 + [\,3\quad 4\,]$$

$$[\,\overbrace{1}^{\text{head}}\quad \overbrace{2\quad 3\quad 4}^{\text{tail}}\,]$$

$$1 + [\,2\quad 3\quad 4\,]$$

$$2 + [\,3\quad 4\,]$$

$$3 + [\,4\,]$$

$$[\overbrace{1}^{\text{head}}\quad \overbrace{2\quad 3\quad 4}^{\text{tail}}]$$

1 + [2 3 4]

2 + [3 4]

3 + [4]

Sum is zero

4 + [ ]

Honoring Immutability

1. Extracting a new list from the old list
2. Calculating a new total

sum ([1, 2, 3, 4])

1 + sum ([2, 3, 4])

1 + (2 + sum ([3, 4]))

1 + (2 + (3 + sum ([4])))

1 + (2 + (3 + (4 + sum ([ ]))))

1 + (2 + (3 + (4 + 0)))

1 + (2 + (3 + 4))

1 + (2 + 7)

1 + 9

10

sum ([1, 2, 3, 4])

1 + sum ([2, 3, 4])

1 + (2 + sum ([3, 4]))

1 + (2 + (3 + sum ([4])))

1 + (2 + (3 + (4 + sum ([ ]))))

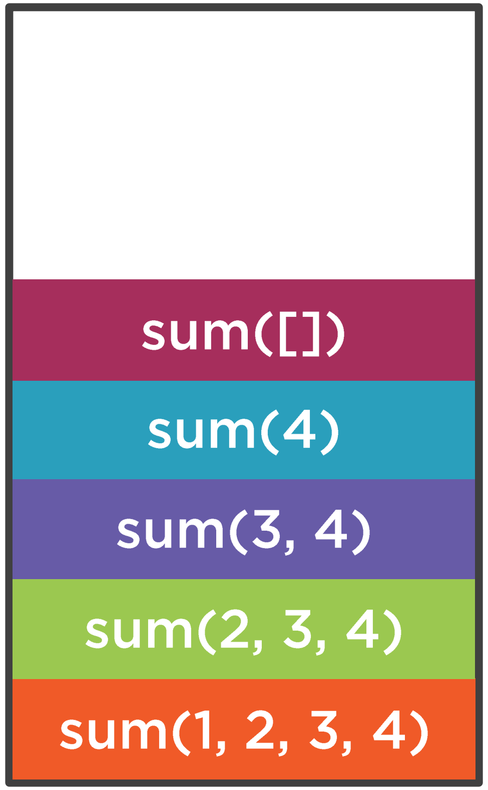
1 + (2 + (3 + (4 + 0)))

1 + (2 + (3 + 4))

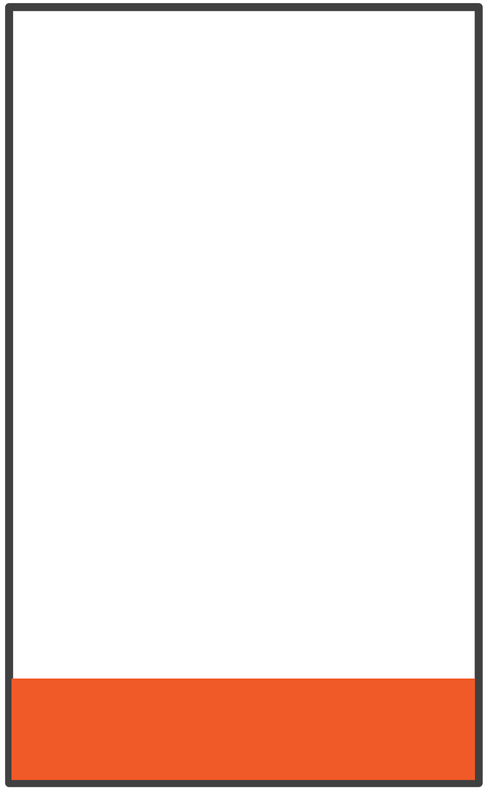
1 + (2 + 7)

1 + 9

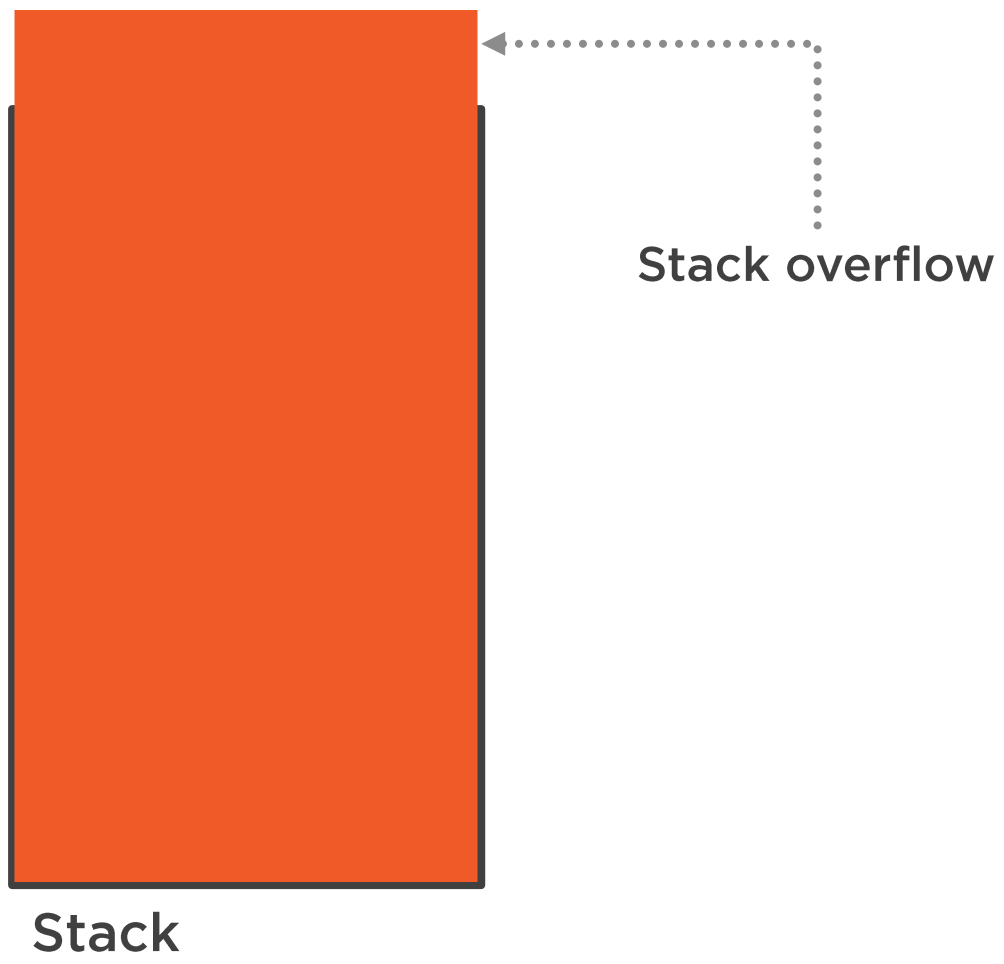
10



Stack



Stack


Stack overflow
Stack

Tail Recursion

```java
int recursiveMethod(int i) {
    // ...
    return recursiveMethod(j);
}
```

Nothing to execute

Tail Recursion

```
public static void main(String args[]) {
    // ...
    int result = recursiveMethod(3);
    // ...
}

int recursiveMethod(int i) {
    // ...
}
```

Tail-Recursive Function

sum ([1, 2, 3, 4], 0)

sum ([2, 3, 4], 1)

sum ([3, 4], 3)

sum ([4], 6)

sum ([ ], 10)

10

With and Without Tail Recursion

With tail recursion

sum ([1, 2, 3, 4], 0)

sum ([2, 3, 4], 1)

sum ([3, 4], 3)

sum ([4], 6)

sum ([ ], 10)

10

Without tail recursion

sum ([1, 2, 3, 4])

1 + sum ([2, 3, 4])

1 + (2 + sum ([3, 4]))

1 + (2 + (3 + sum ([4])))

1 + (2 + (3 + (4 + sum ([ ]))))

1 + (2 + (3 + (4 + 0)))

1 + (2 + (3 + 4))

1 + (2 + 7)

1 + 9

10

To Create a Tail-recursive Function

Create a private recursive function
with an additional accumulator parameter

The base case of the recursive
function returns the accumulator

The recursive invocation provides
an updated value for the accumulator

Create a public function that calls the
tail-recursive function using the appropriate initial values

In Summary

Non-tail recursive functions will use the stack to remember the state

Tail recursive functions use accumulators to remember state

Tail Call Optimization (TCO)

When the compiler automatically make tail-recursive functions more efficient.

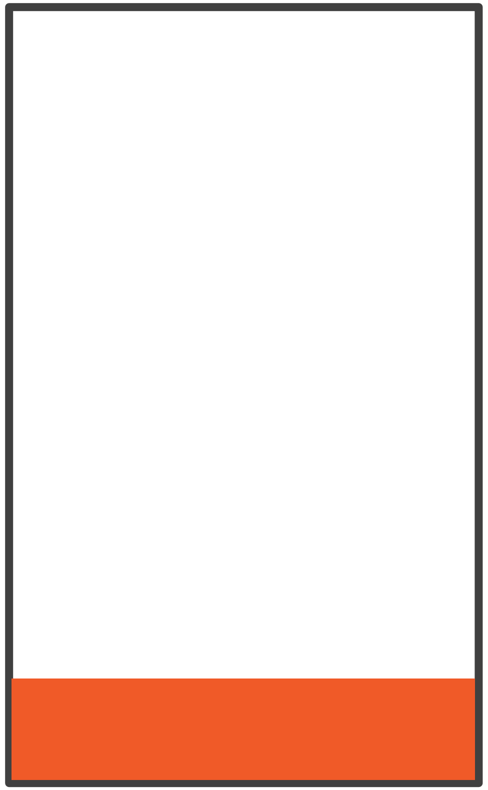
Tail Call Optimization with Trampolines

Thunk

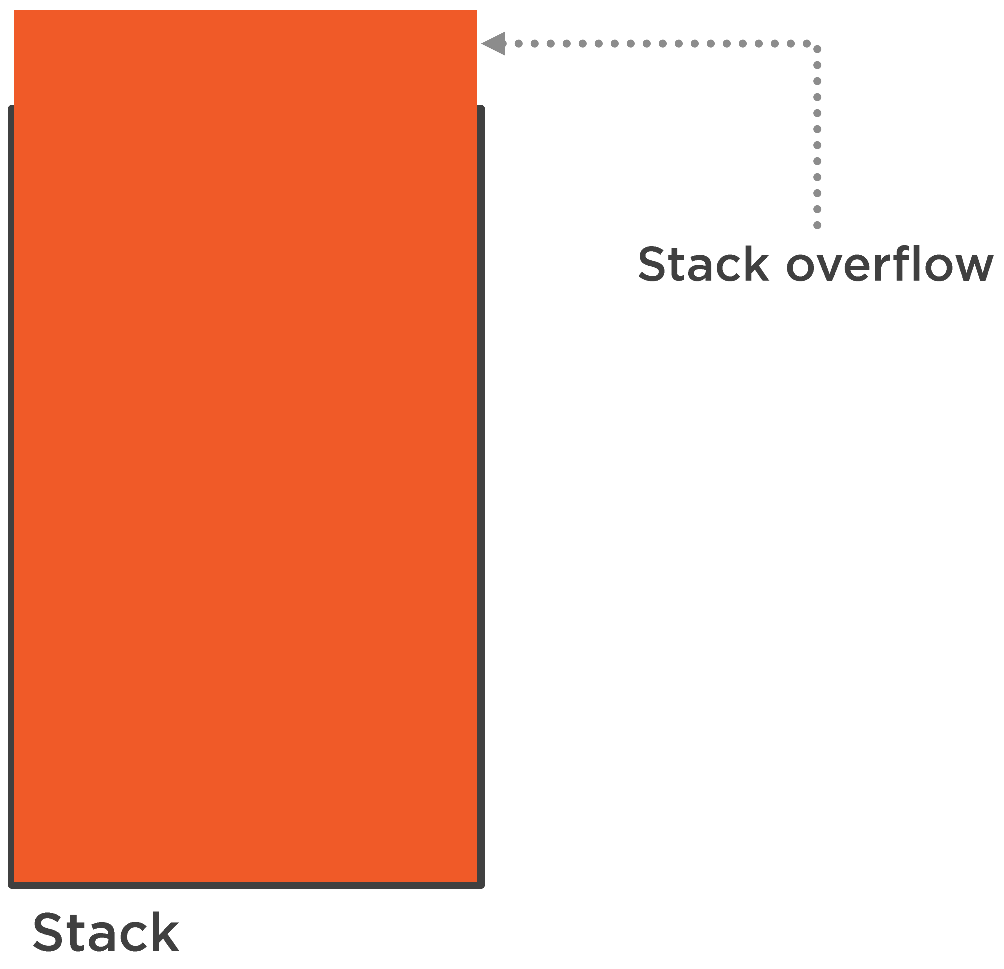
A function that is returned by another function to delay a computation.

Implemented with the Supplier Interface

```java
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

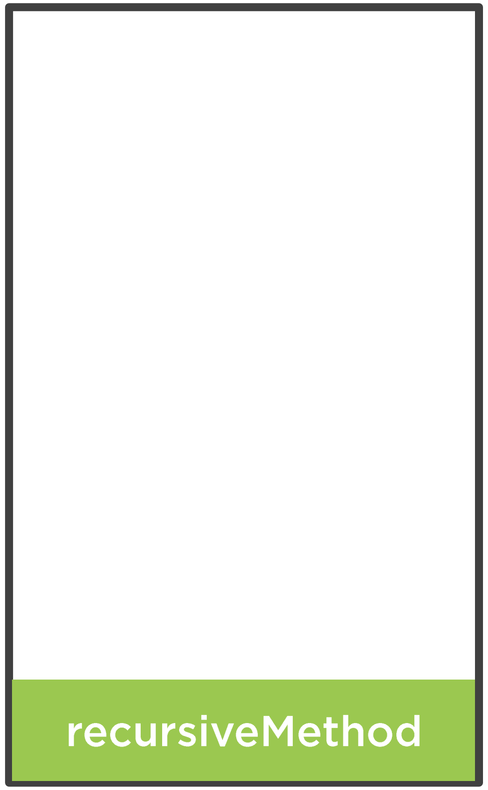


Stack


Stack overflow
Stack

```java
int recursiveMethod(int i) {
    // ...
    return () -> recursiveMethod(j);
}
```

```java
int recursiveMethod(int i) {
    // ...
    return () -> recursiveMethod(j);
}
```



Stack

Trampoline

A loop that iteratively invokes a function that can return a thunk.

Trampoline Pseudocode

```
while (f == 'function') {
    f = f();
}

return f;
```

The Fold Operation

```java
Integer sum(List<Integer> list, int acc) {
  return list.isEmpty()
      ? acc
      : sum( tail(list), acc + head(list) );
}
```

```java
Integer sum(List<Integer> list, int acc) {
  return list.isEmpty()
      ? acc
      : sum( tail(list), acc + 1 );
}
```

```java
Integer length(List<Integer> list, int acc) {
  return list.isEmpty()
      ? acc
      : length( tail(list), acc + 1 );
}
```

```java
Integer length(List<Integer> list, List<Integer> acc) {
  return list.isEmpty()
      ? acc
      : length( tail(list), acc + 1 );
}
```

```java
List<Integer> length(List<Integer> list, List<Integer> acc) {
  return list.isEmpty()
      ? acc
      : length( tail(list), acc + 1 );
}
```

```java
List<Integer> length(List<Integer> list, List<Integer> acc) {
  return list.isEmpty()
      ? acc
      : length( tail(list), concat(Arrays.asList(head(list)), acc));
}
```

```java
List<Integer> reverse(List<Integer> list, List<Integer> acc) {
  return list.isEmpty()
      ? acc
      : reverse( tail(list), concat(Arrays.asList(head(list)), acc));
}
```

Higher-level of Abstraction

```
T function(List<T> list, T acc, Function op) {
   return list.isEmpty()
      ? acc
      : function(tail(list), op.apply(...));
}
```

The Function Takes Two Values

The head of the list

The accumulator

Type of the Function

$$(T, U) \rightarrow U$$

Type of the Function (Curried Version)

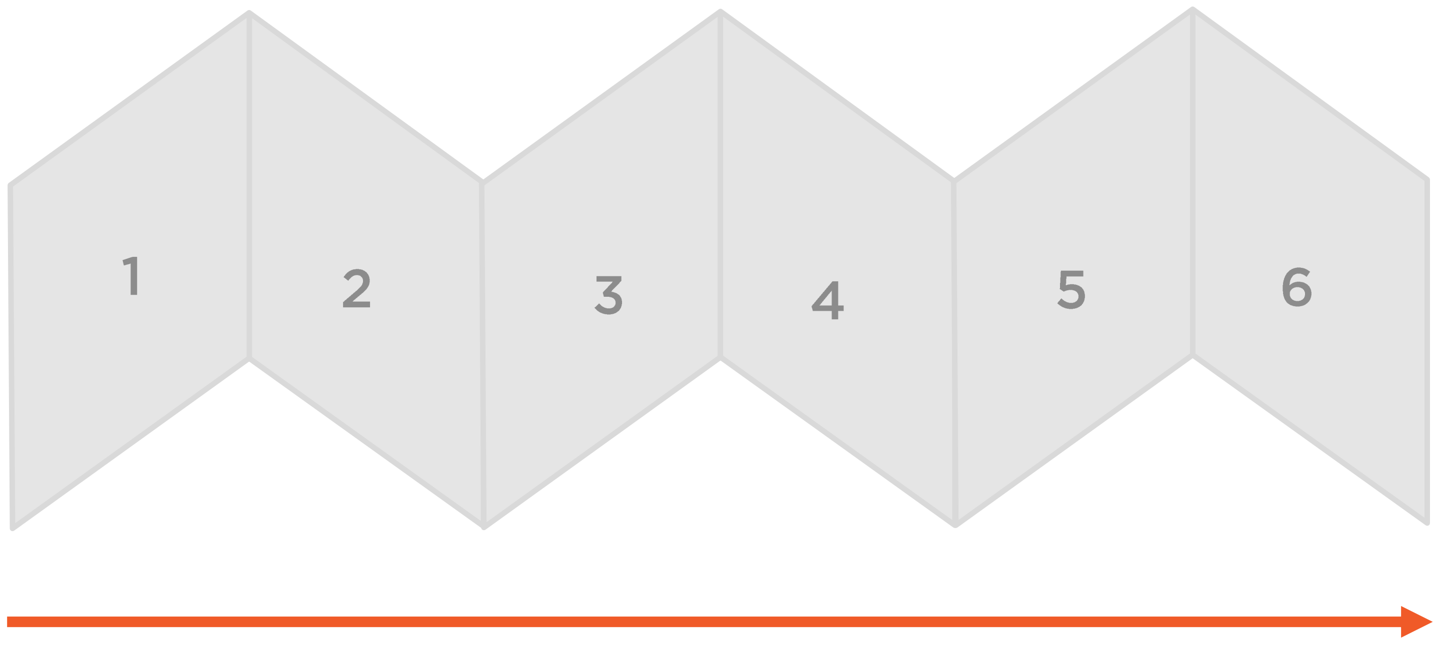
$$T \rightarrow U \rightarrow U$$

```java
Function<Integer, Function<Integer, Integer>> sum = x -> y -> x + y;

Integer generic(List<Integer> list,
                int acc,
                Function<Integer, Function<Integer, Integer>> f) {
    return list.isEmpty()
           ? acc
           : generic(tail(list), f.apply(head(list)).apply(acc), f);
}
```

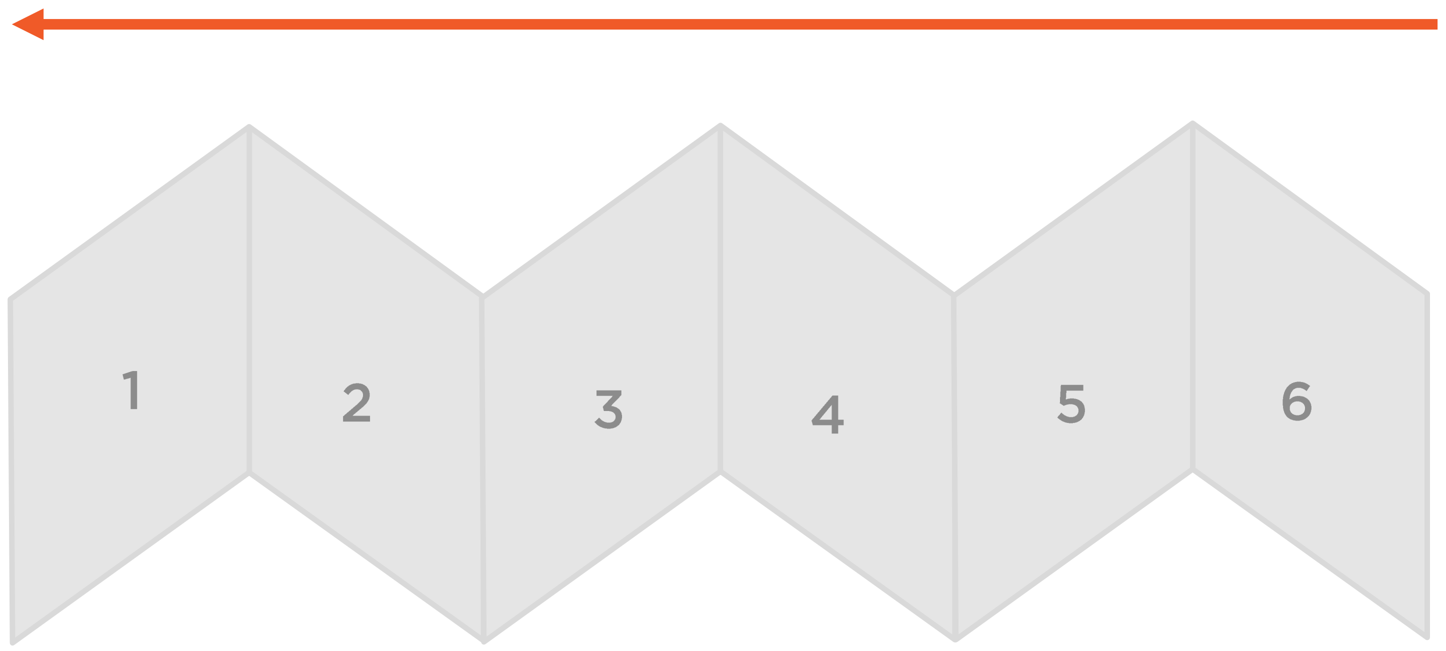
```java
Function<Integer, Function<Integer, Integer>> sum = x -> y -> x + y;

Integer foldLeft(List<Integer> list,
                 int acc,
                 Function<Integer, Function<Integer, Integer>> f) {
  return list.isEmpty()
         ? acc
         : foldLeft(tail(list), f.apply(head(list)).apply(acc), f);
}
```



foldLeft

foldRight



Folding in Two Directions

If the operation is commutative, both ways of folding are equivalent

If the operation is not commutative, the two ways of folding give different results

Is Stream.reduce()
equivalent to foldLeft?

Folding Versus Mapping, Reducing, and Collecting

Is Stream.reduce()
equivalent to foldLeft?

NO

Reducing

Folding to a result that is the same type as the list elements.

FoldLeft

```
U foldLeft(List<T> ts,
                U identity,
                Function<U, Function<T, U>> f) {
    // ...
}
```

First Version of Reduce

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

First Version of Reduce

```
Optional<T> reduce(BinaryOperator<T> accumulator) {
    boolean foundAny = false;
    T result = null;
    for (T element : this stream) {
        if (!foundAny) {
            foundAny = true;
            result = element;
        }
        else
            result = accumulator.apply(result, element);
    }
    return foundAny ? Optional.of(result) : Optional.empty();
}
```

Second Version of Reduce

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

Second Version of Reduce

```
T reduce(T identity, BinaryOperator<T> accumulator) {
    T result = identity;
    for (T element : this stream)
        result = accumulator.apply(result, element)
    return result;
}
```

Third Version of Reduce

```
U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)
```

Third Version of Reduce

```
U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner) {
    U result = identity;
    for (T element : this stream)
        result = accumulator.apply(result, element)
    return result;
}
```

Collect

```
R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
{
     R result = supplier.get();
     for (T element : this stream)
         accumulator.accept(result, element);
     return result;
}
```

Reduce is designed to work with immutable objects, while collect can only work with mutable objects.

Memoization

Memoization

A technique that caches the result of an operation so it can be returned immediately if the same computation is performed in the future.

Memoization Optimizes Execution Time

$$O(n) \rightarrow O(1)$$

Memoization Is a Trade

Reduces execution time

Increases memory

Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8 ...

$$F_n = F_{n-1} + F_{n-2}$$

Except for the first two numbers that are always 0 and 1.

Fib(4)

fib(4)

fib(3) + fib(2)

fib(2) + fib(1) + fib(1) + fib(0)

fib(1) + fib(0)

n	Fib(n)	Number of calls
0	1	1
1	1	1
2	2	3
3	3	5
4	5	9
5	8	15
6	13	25
7	21	41
8	34	67
9	55	109
10	89	177
11	144	287
12	233	465
13	377	753
14	610	1219
15	987	1973

Memoization is often an alternative to tail call optimization to improve performance.

Memoization Is Compatible with Functional Programming

A memoized function always returns the same value for the same argument

If the side effect of storing the results is not visible from outside the function

```java
public Double average(int number) {
   return IntStream.rangeClosed(1, number).average().orElse(0.0);
}

// ...

Function<Integer, Double> avg = this::average;
```

```java
public Double average(int number) {
   return IntStream.rangeClosed(1, number).average().orElse(0.0);
}

// ...

Map<Integer, Double> cache = new ConcurrentHashMap<>();
Function<Integer, Double> avg =
                    x -> cache.computeIfAbsent(x, this::average);
```

```java
class Functions {
    private static Map<Integer, Double> cache = new ConcurrentHashMap<>();
    public static Function<Integer, Double> avg =
                  x -> cache.computeIfAbsent(x, Functions::average);

    private static Double average(int number) {
        return IntStream.rangeClosed(1, number).average().orElse(0.0);
    }
}
```

Things to Remember

For simple use cases, you can use the Stream API

For more advanced use cases, you must implement your own types

Things to Remember

We can replace loops with recursive functions

Be careful with stack overflow exceptions

Write tail-recursive functions, where the recursive call is the last line of the function

- Tail-recursive functions use an accumulator to carry intermediate state
- Tail call optimization (TCO)
- Implement it with thunks and trampolines

Things to Remember

Fold function

- Tail-recursive function for performing operations over collections
- Two directions, from left to right or from right to left
- If the operation is commutative, both ways of folding are equivalent
- Java doesn't have an equivalent of this function

Things to Remember

Memoization

- Caching the result of an operation so it can be returned immediately if the same computation is performed in the future

Using imperative structures inside a pure function is not bad if the function remains pure

In the Next Module

Dealing with nulls functionally