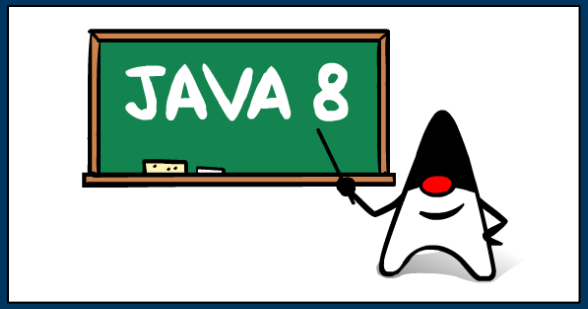
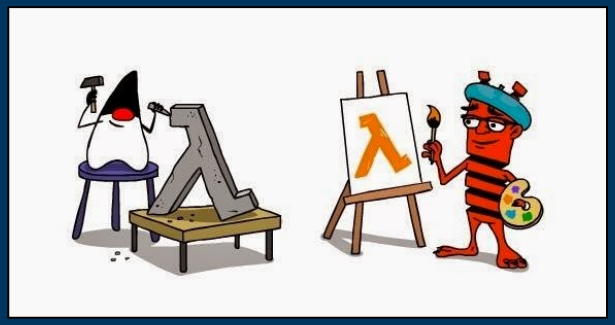
2nd Expedia Tech 'Know How' Talk (18 Nov 2015)

Lambda Expressions and Java 8

Lambda Calculus, Lambda Expressions, Syntactic Sugar, First Class Functions





Expedia Tech
KNOW HOW

USE OUR HASHTAG
#expediaeng

Official twitter account for Expedia Worldwide Engineering

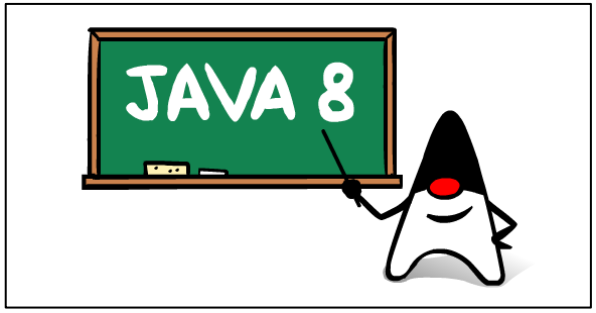
https://twitter.com/ExpediaEng

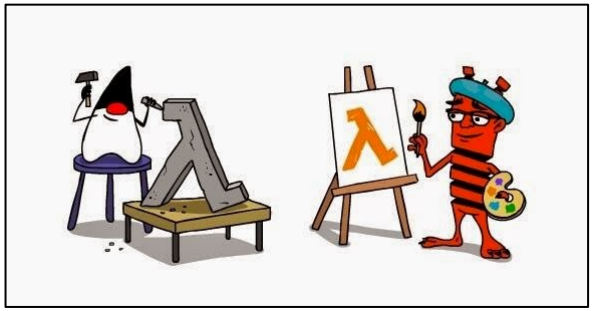
Expedia
Engineering @Expedia

https://techblog.expedia.com/

Lambda Expressions and Java 8

Lambda Calculus, Lambda Expressions, Syntactic Sugar, First Class Functions





https://twitter.com/philip_schwarz
http://www.slideshare.net/pjschwarz
philip.johann.schwarz@gmail.com

Some of the examples in this talk will be entered in the Java REPL,
a simple Read-Eval-Print-Loop for the Java language

The REPL can be used online @ http://www.javarepl.com/console.html

Java REPL

```
Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0 on Linux 3.5.0-21-generic
Welcome to JavaREPL Web Console version 272

java> 1 + 1
java.lang.Integer res0 = 2
java> int x = 1
int x = 1
java> String hello(String name){
java>     return "Hello," + name + "!";
java> }
Created method java.lang.String hello(java.lang.String)
java> hello("John");
java.lang.String res2 = "Hello,John!"
java>
```

It can also be downloaded as a JAR and started as follows:

```
$ java -jar javarepl.jar
```

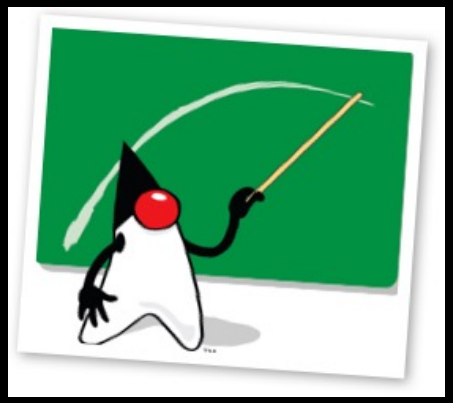
```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
```

```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
```

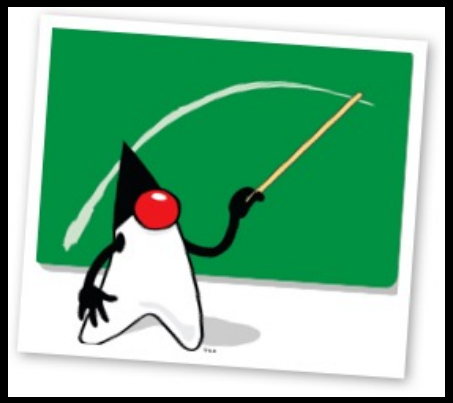
Hello and welcome to this tutorial on Java 8 Lambdas and Streams

```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
```

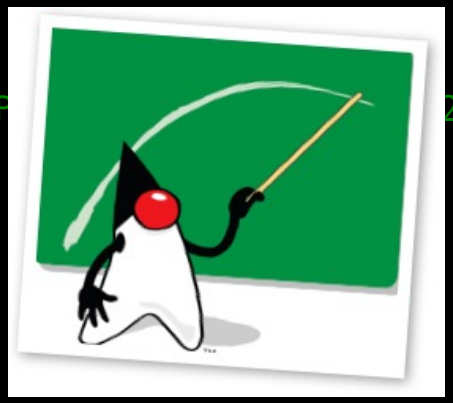
Let's get started by creating a stream of numbers



```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java>
```

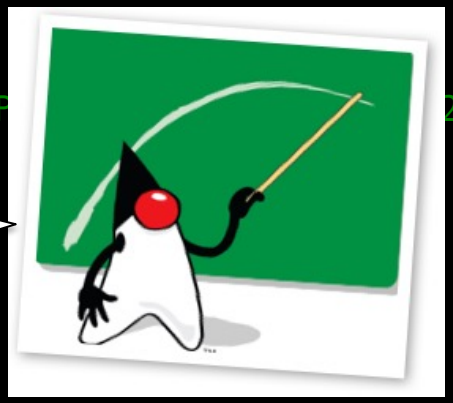


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                    2d
java>
```

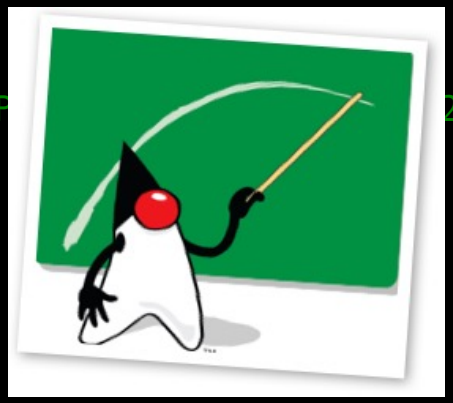


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1
java.util.stream.Stream stream
java>
```

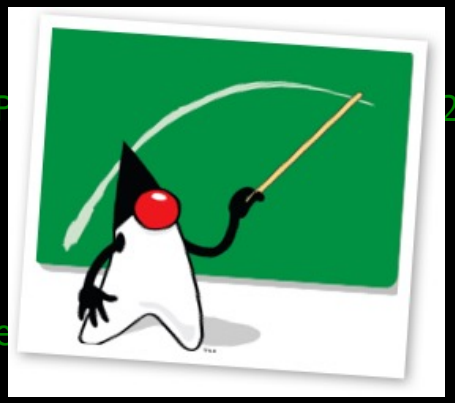
Actually, to keep things simple, let's create the **primitive int specialisation** of **Stream**



```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java>
```

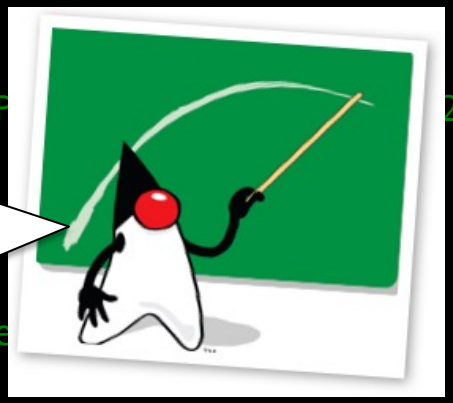


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2, 3)
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
```

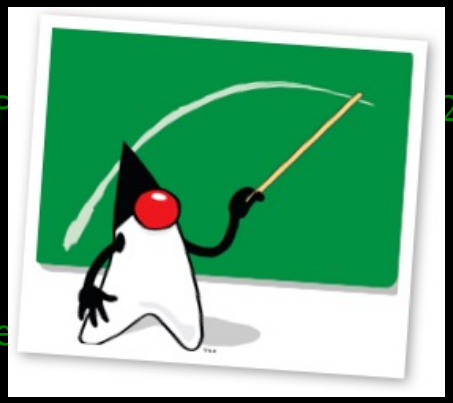


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1,
java.util.stream.IntStream stream = java.
java>
```

What are the zero-args instance methods of **IntStream**?

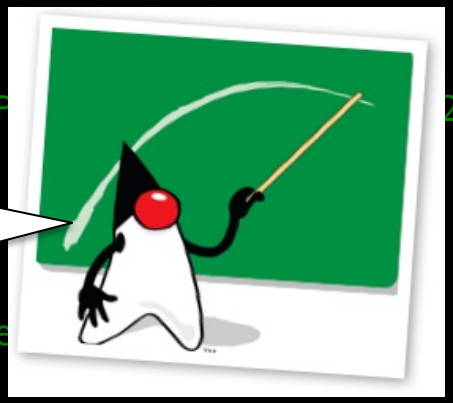


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2, 3)
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
```

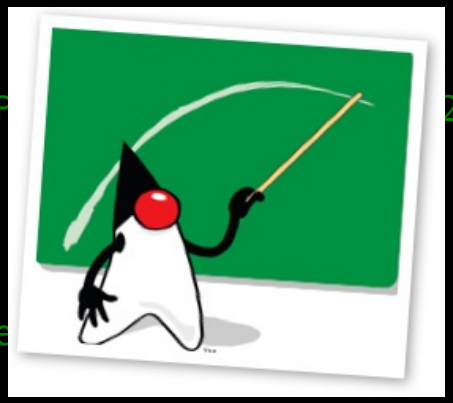


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceP                    2d
java>
java> import java.util.stream.IntStream
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1,
java.util.stream.IntStream stream = java.ut                    IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
```

Let's start with the simplest one: **count**!

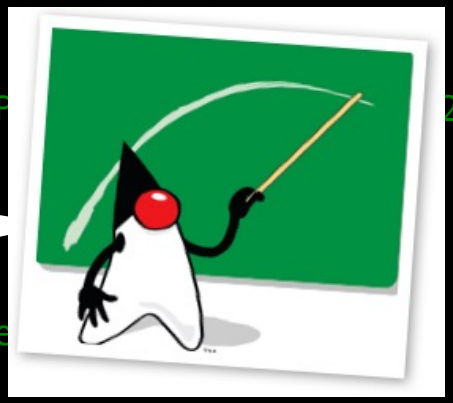


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                    2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2, 3)
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
```

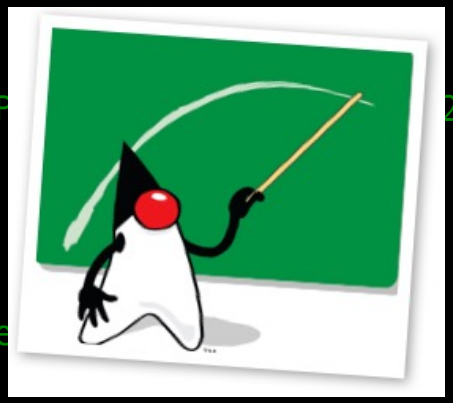


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                    2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2,
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
```

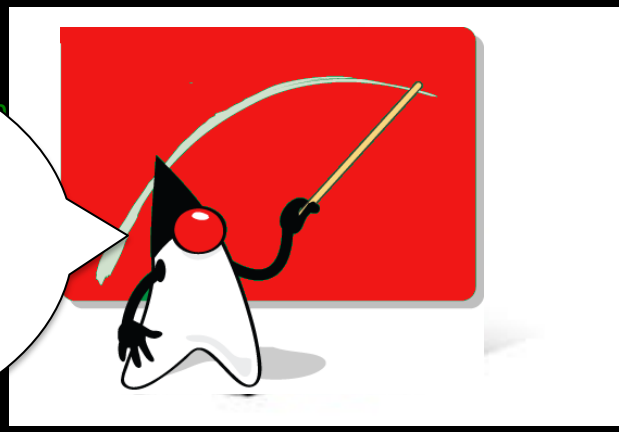
Now try **sum**



```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                    2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2, 3)
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
java> stream.sum()
java.lang.IllegalStateException: stream has already been operated upon or closed
java>
```

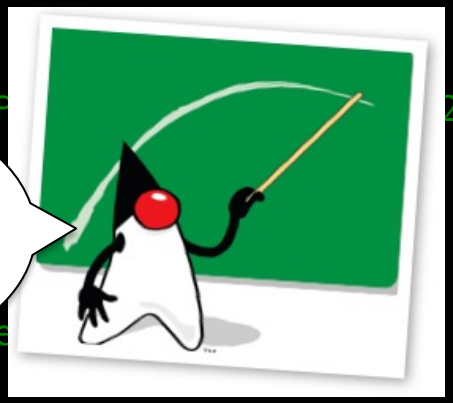


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java
java>
java> import java.util.stream.IntS
Imported java.util.stream.IntStrea
java> IntStream stream = IntStream
java.util.stream.IntStream stream =
java>
java> show_no_args_instance_methods_of(Int        class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
java> stream.sum()
java.lang.IllegalStateException: stream has already been operated upon or closed
java>
```

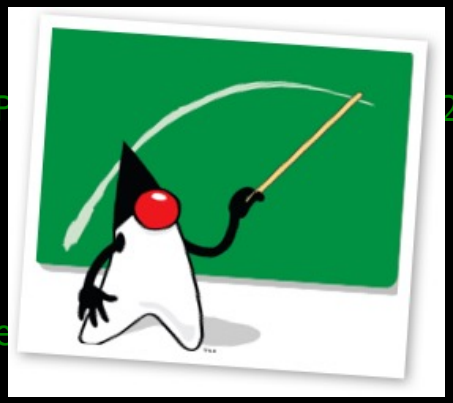


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stre                    2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2
java.util.stream.IntStream stream = java.uti
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
java> stream.sum()
java.lang.IllegalStateException: stream has already been operated upon or closed
java>
```

Now try summing a fresh stream

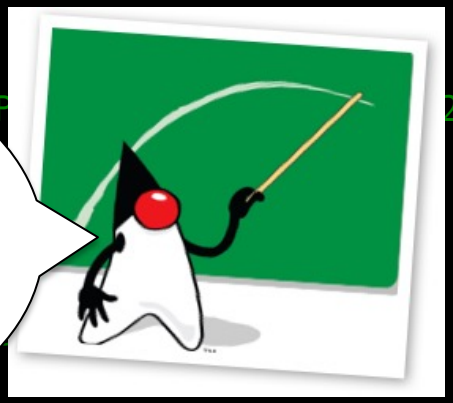


```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.util.stream.ReferenceF                    2d
java>
java> import java.util.stream.IntStream;
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.of(1, 2, 3)
java.util.stream.IntStream stream = java.util.stream.IntPipe
java>
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
java> stream.sum()
java.lang.IllegalStateException: stream has already been operated upon or closed
java>
java> IntStream.of(1, 2, 3).sum()
java.lang.Integer res4 = 6
java>
```



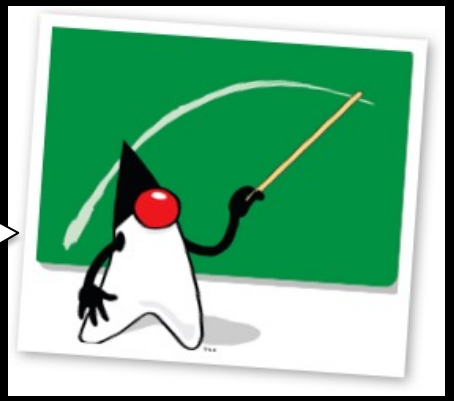
```
$ java -jar javarepl.jar
Welcome to JavaREPL version 272 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25)
Type expression to evaluate, :help for more options or press tab to auto-complete.
java>
java> import java.util.stream.Stream;
Imported java.util.stream.IntStream
java> Stream stream = Stream.of(1, 2, 3)
java.util.stream.Stream stream = java.
java>
java> import java.util.stream.IntSt
Imported java.util.stream.IntStream
java> IntStream stream = IntStream.
java.util.stream.IntStream stream =
java>
java> show_no_args_instance_methods_of(Int
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java>
java> stream.count()
java.lang.Long res3 = 3
java>
java> stream.sum()
java.lang.IllegalStateException: stream has already been operated upon or closed
java>
java> IntStream.of(1, 2, 3).sum()
java.lang.Integer res4 = 6
java>
```

Before we can use **min** and **max** we need to introduce **Optional**



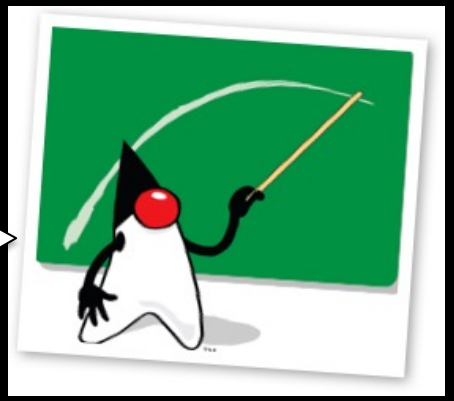
java>

Actually, to keep things simpler, let's look at the **int primitive specialisation** of **Optional**



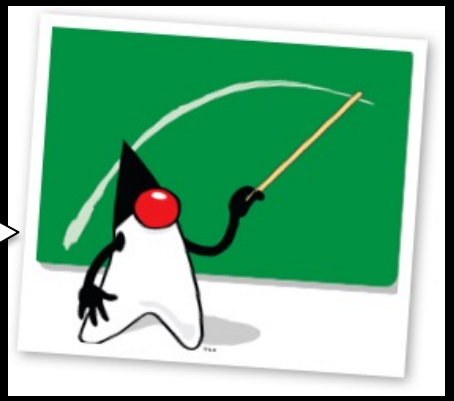
java>

OptionalInt is a container object which may or may not contain a int value

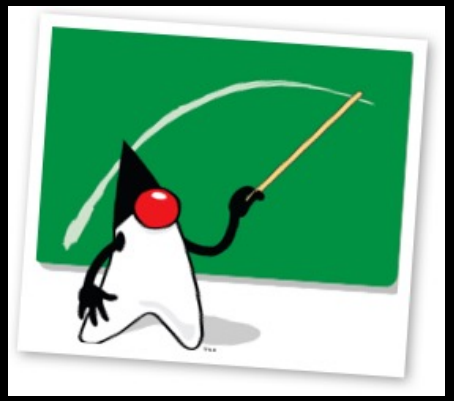


java>

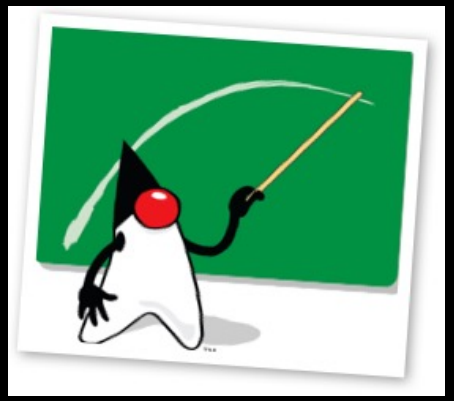
Let’s create both an **OptionalInt** containing 10 and an empty **OptionalInt**



```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java>
```

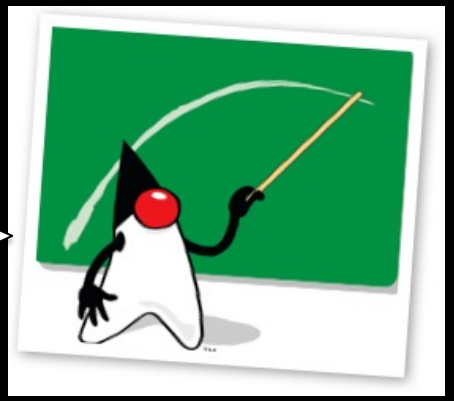


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java>
```

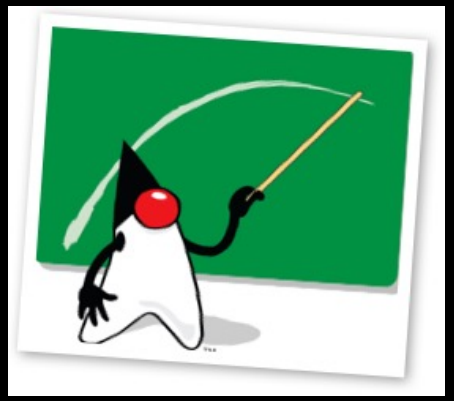


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java>
```

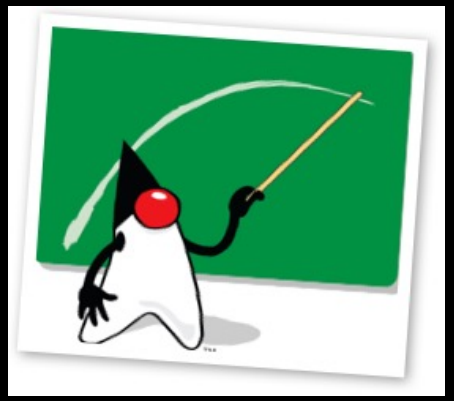
Now let's ask both optionals whether they contain a value



```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java>
```

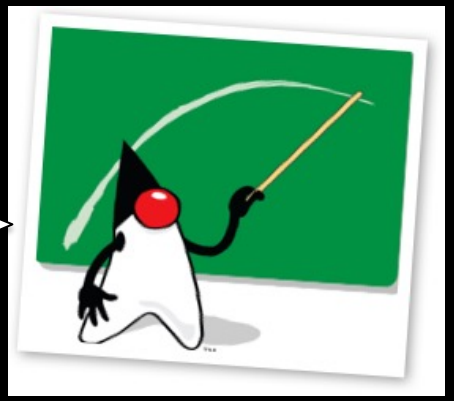


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
```

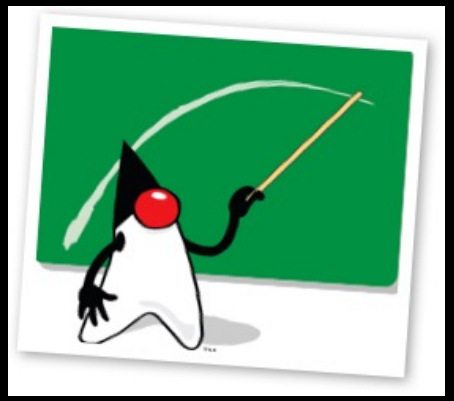


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
```

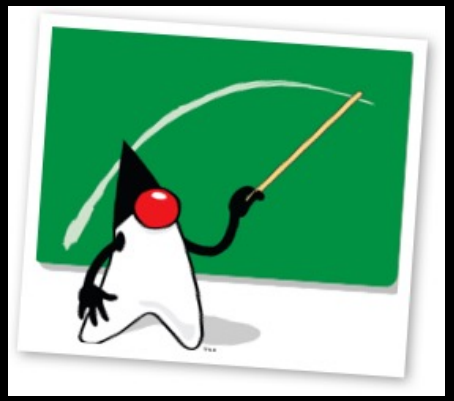
Now let's ask both optionals for their value



```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java
```

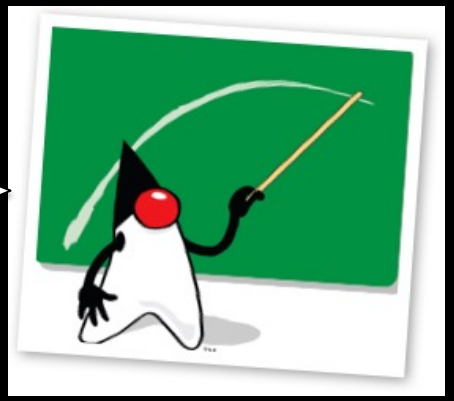


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
```



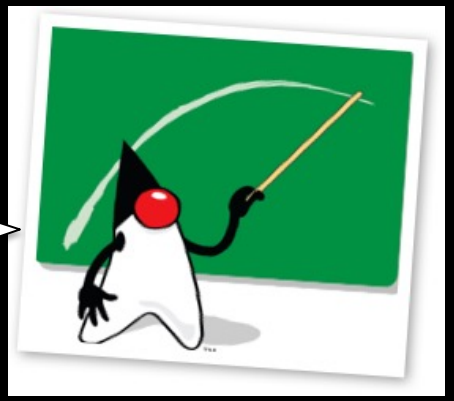
```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
```

Remember to call **isPresent**() before calling **getAsInt**()!

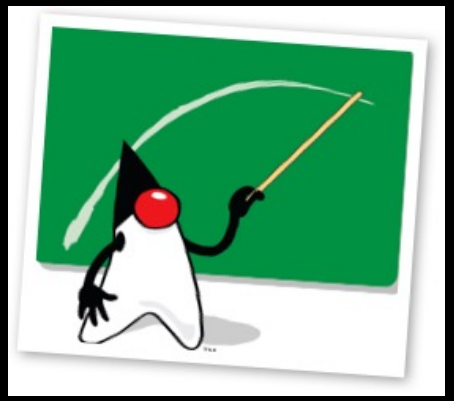


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
```

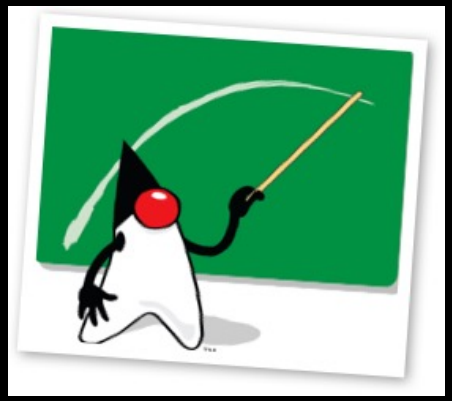
Alternatively, use **orElse**()



```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java>
```

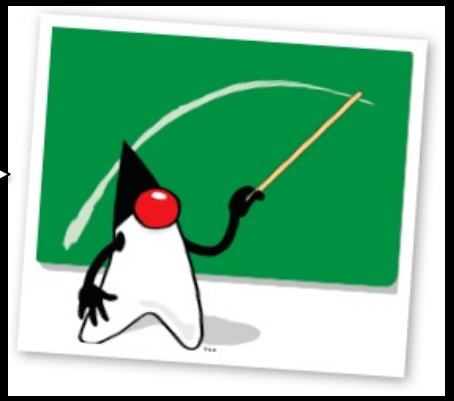


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
```

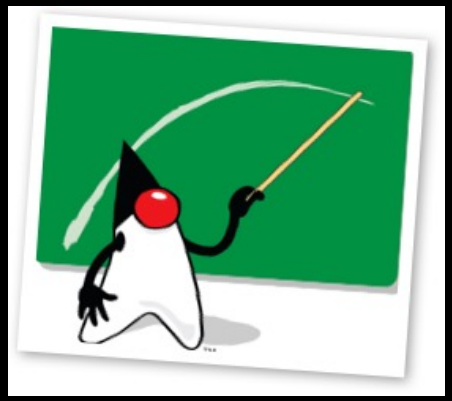


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
```

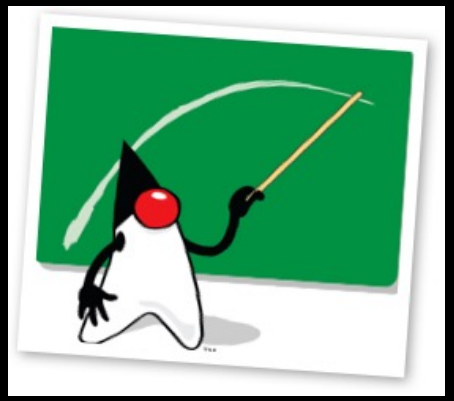
Now we can look at the **min** and **max** methods of **IntStream**



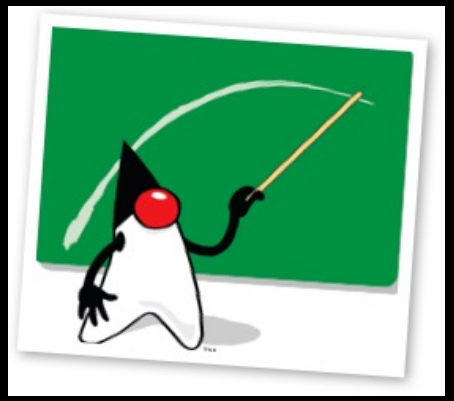
```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
java> IntStream.empty().max()
java.util.OptionalInt res18 = OptionalInt.empty
java>
```



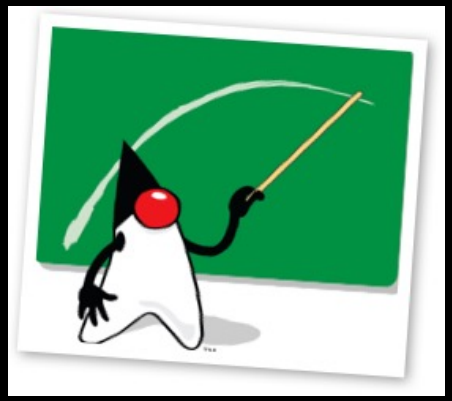
```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
java> IntStream.empty().max()
java.util.OptionalInt res18 = OptionalInt.empty
java> IntStream.of(1, 2, 3).max()
java.util.OptionalInt res19 = OptionalInt[3]
java>
```



```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
java> IntStream.empty().max()
java.util.OptionalInt res18 = OptionalInt.empty
java> IntStream.of(1, 2, 3).max()
java.util.OptionalInt res19 = OptionalInt[3]
java> IntStream.of(1, 2, 3).max().getAsInt()
java.lang.Integer res20 = 3
java>
```

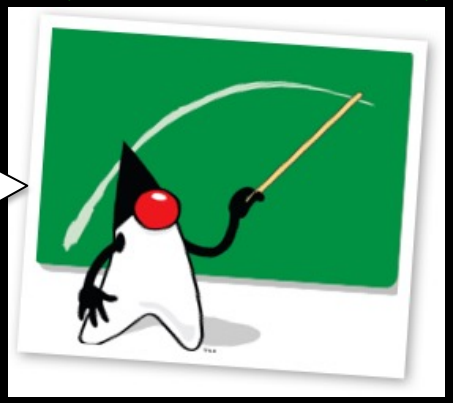


```
java>
java> OptionalInt fullOp = OptionalInt.of(10)
java.util.OptionalInt fullOp = OptionalInt[10]
java> OptionalInt emptyOp = OptionalInt.empty()
java.util.OptionalInt emptyOp = OptionalInt.empty
java> fullOp.isPresent()
java.lang.Boolean res13 = true
java> emptyOp.isPresent()
java.lang.Boolean res14 = false
java>
java> fullOp.getAsInt()
java.lang.Integer res15 = 10
java> emptyOp.getAsInt()
java.util.NoSuchElementException:
No value present
java>
java> emptyOp.orElse(5)
java.lang.Integer res16 = 5
java> fullOp.orElse(5)
java.lang.Integer res17 = 10
java>
java> IntStream.empty().max()
java.util.OptionalInt res18 = OptionalInt.empty
java> IntStream.of(1, 2, 3).max()
java.util.OptionalInt res19 = OptionalInt[3]
java> IntStream.of(1, 2, 3).max().getAsInt()
java.lang.Integer res20 = 3
java> IntStream.of(1, 2, 3).min().getAsInt()
java.lang.Integer res21 = 1
java>
```

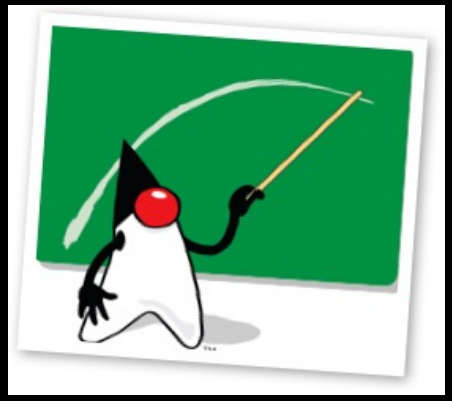


```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator,  asLongStream, asDoubleStream,
summaryStatistics]
java>
```

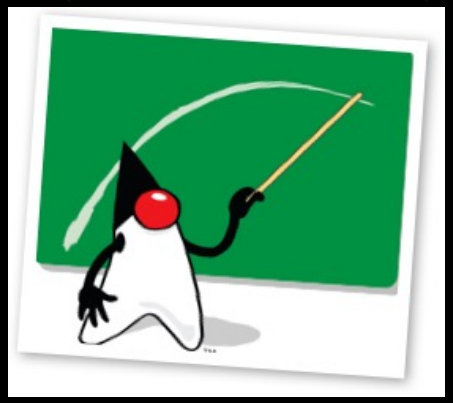
We've seen **min**, **max**, **sum**, **count**. Let's look at two closely related methods: **average** & **summaryStatistics**



```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).average()
java.util.OptionalDouble res17 = OptionalDouble[2.0]
java>
```

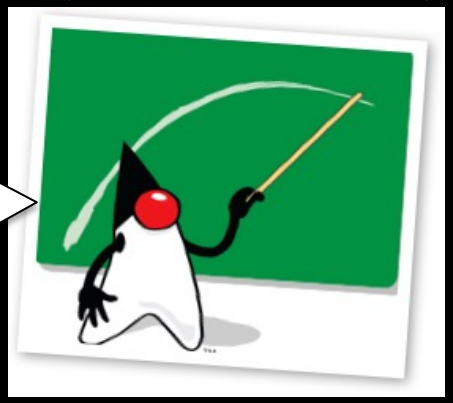


```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).average()
java.util.OptionalDouble res17 = OptionalDouble[2.0]
java> IntStream.of(1, 2, 3).summaryStatistics()
java.util.IntSummaryStatistics res18 = IntSummaryStatistics
{count=3, sum=6, min=1, average=2.000000, max=3}
java>
```



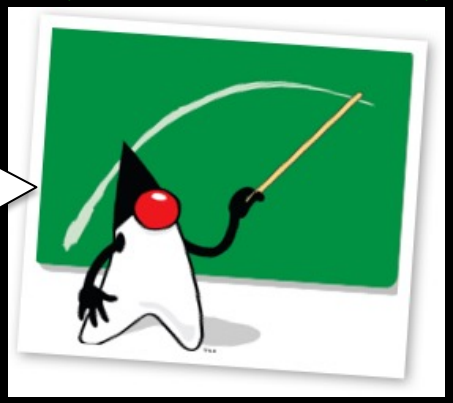
```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).aver
java.util.OptionalDouble res17
java> IntStream.of(1, 2, 3).s
java.util.IntSummaryStatistic
{count=3, sum=6, min=1, averag
java>
```

All the methods seen so far compute some result(s) using the stream's values.



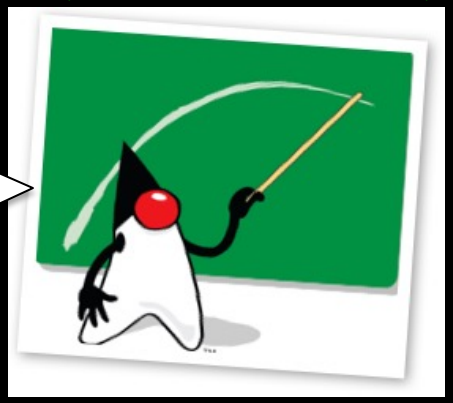
```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator,  asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).ave
java.util.OptionalDouble res17
java> IntStream.of(1, 2, 3).s
java.util.IntSummaryStatistics
{count=3, sum=6, min=1, average
java>
```

How can we use the stream's values to **DO** something, rather than **COMPUTE** something?



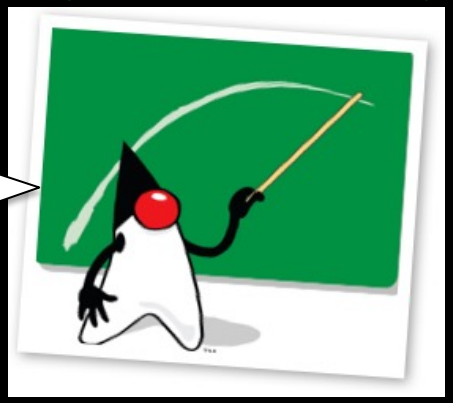
```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).average()
java.util.OptionalDouble res17 = Opt
java> IntStream.of(1, 2, 3).summary
java.util.IntSummaryStatistics res18
{count=3, sum=6, min=1, average=2.000000
java>
```

E.g. how do we print the stream's values to the console?



```
java> show_no_args_instance_methods_of(IntStream.class)
[min, max, sum, count, boxed, close, sorted, toArray, average, findAny, iterator,
iterator, parallel, parallel, distinct, findFirst, unordered, sequential,
sequential, isParallel, spliterator, spliterator, asLongStream, asDoubleStream,
summaryStatistics]
java> IntStream.of(1, 2, 3).average()
java.util.OptionalDouble res17 = O
java> IntStream.of(1, 2, 3).summa
java.util.IntSummaryStatistics res
{count=3, sum=6, min=1, average=2.000000,
java>
```

We can use **IntStream's forEach** method



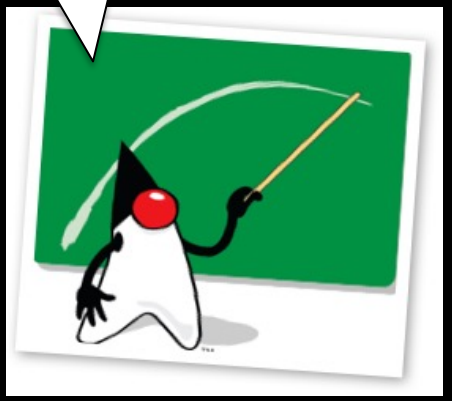
```
java>
```

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

Here is the Javadoc for **IntStream**'s **forEach** method



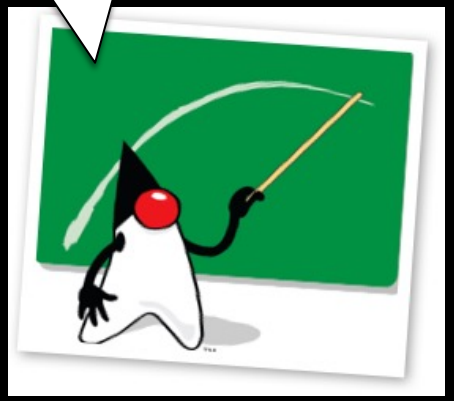
java>

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

The argument of **forEach** implements interface **IntConsumer**



```
java>
```

forEach

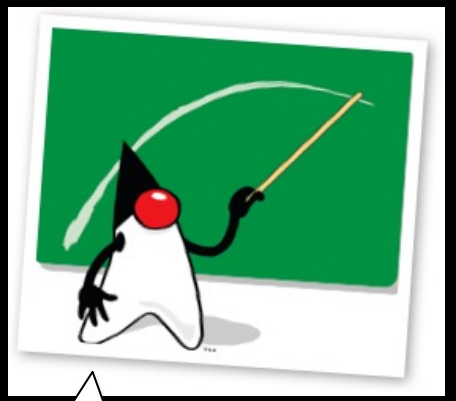
```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

accept

```
void accept(int value)
```

Performs this operation on the given argument.



IntConsumer has one method, called **accept**. Here is its Javadoc

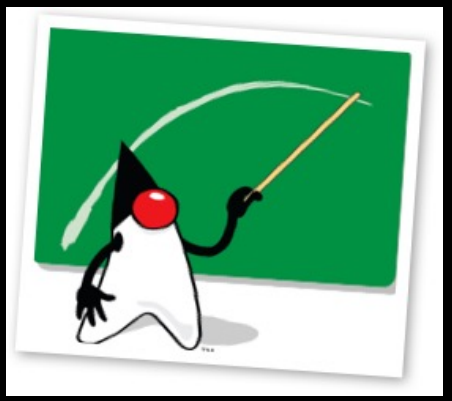
```
java>
```

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach(  ???  )
```



accept

```
void accept(int value)
```

Performs this operation on the given argument.

```
java>
```

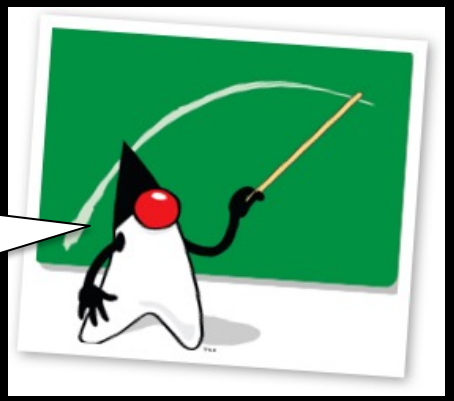
forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach(  ???  )
```

So we need to pass **forEach** an **IntConsumer**



accept

```
void accept(int value)
```

Performs this operation on the given argument.

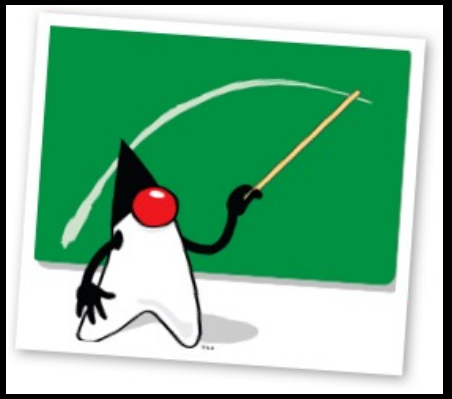
```
java>
```

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
```



accept

```
void accept(int value)
```

Performs this operation on the given argument.

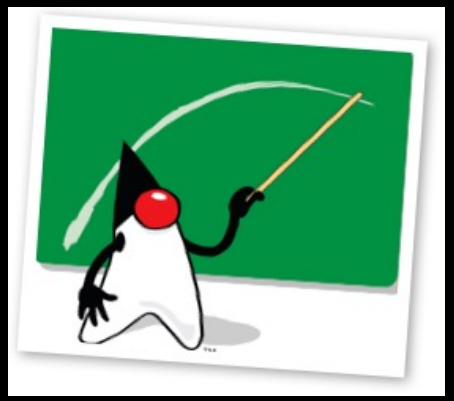
```
java>
```

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
java>
1
2
3
```



accept

```
void accept(int value)
```

Performs this operation on the given argument.

```
java>
```

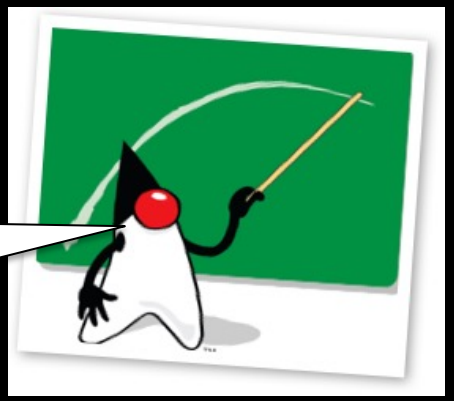
forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
java>
1
2
3
```

forEach called the **IntConsumer's accept** method with each **value** (number) in the stream, which caused the numbers to be printed



accept

```
void accept(int value)
```

Performs this operation on the given argument.

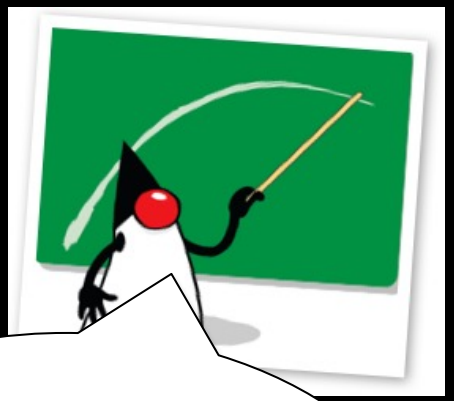
```
java>
```

forEach

```
void forEach(IntConsumer action)
```

Performs an action for each element of this stream.

```
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
java>
1
2
3
```



accept

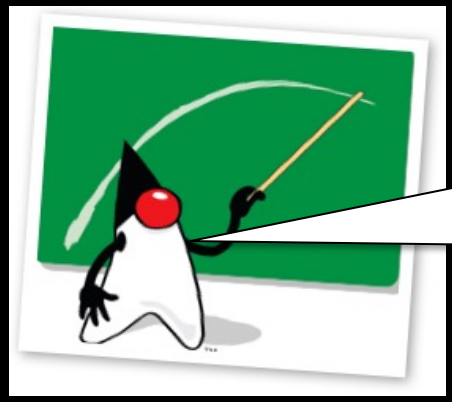
```
void accept(int value)
```

Performs this operation on the given argument.

Now let's see how instead of passing in an anonymous implementation of **IntConsumer**, **Java 8** allows us to pass in something more concise

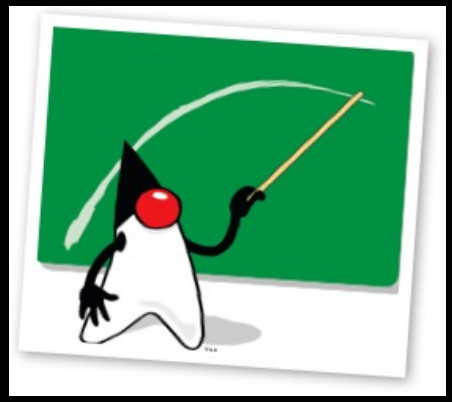
```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
```

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
```

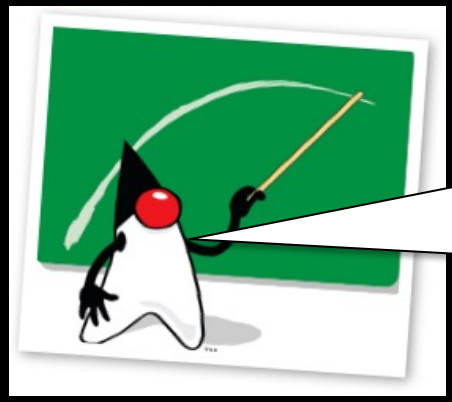


First let's make the code more concise by renaming the parameter from **value** to **n**

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
```

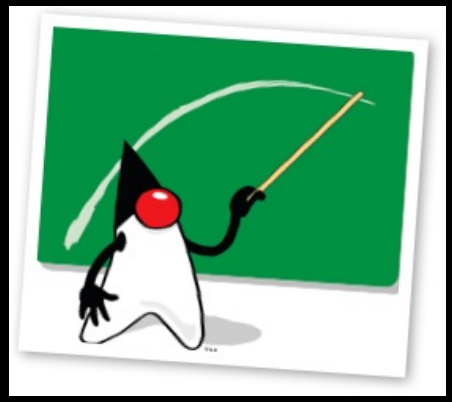


```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
```

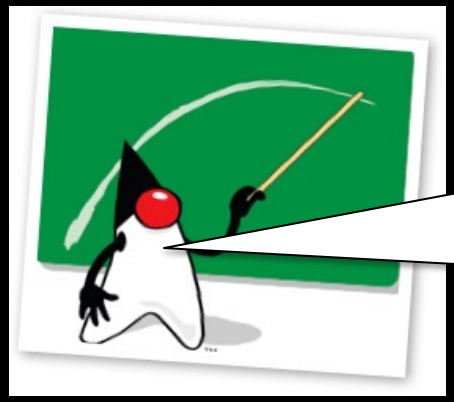


Next, the compiler already knows that we have to pass in an **IntConsumer**, and that this has only one method called **accept**, so we don't need to provide that info, explicitly.

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                      (int n) ->   { System.out.println(n);     }   ,
```

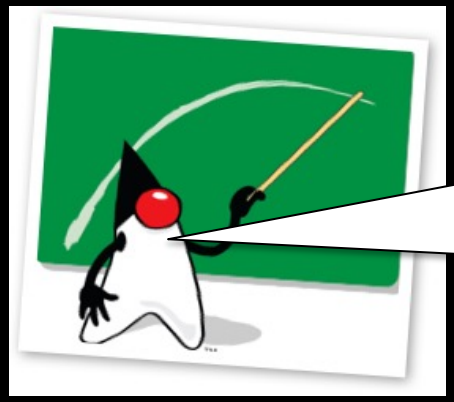


```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                    (int n) ->   { System.out.println(n);     }   ,
```



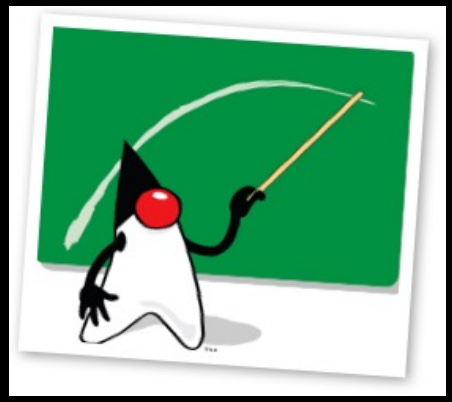
We have replaced the anonymous instance of **IntConsumer** with a **lambda expression** (new in **Java 8**) which consists of a list of formal parameters (one in this case) and a method body, separated by an arrow.

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                    (int n) ->   { System.out.println(n);     }   ,
```

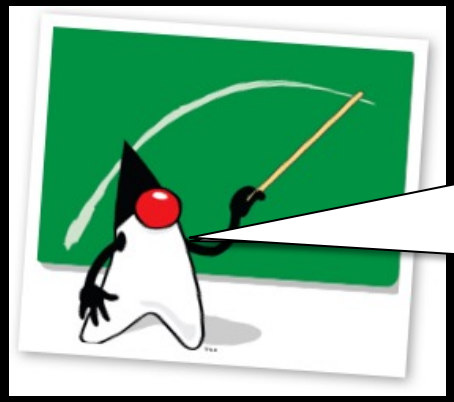


Next, the body of the **accept** method consists of just one statement, so the compiler doesn't need us to wrap the statement in braces: let's remove them

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                      (int n) ->   { System.out.println(n);     }   ,
    |                                      (int n) ->     System.out.println(n);         ,
```

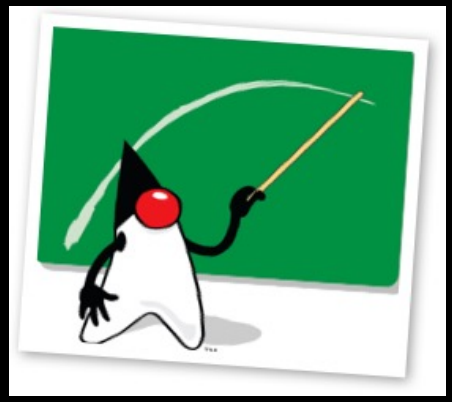


```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                    (int n) ->   { System.out.println(n);     }   ,
    |                                    (int n) ->     System.out.println(n);         ,
```

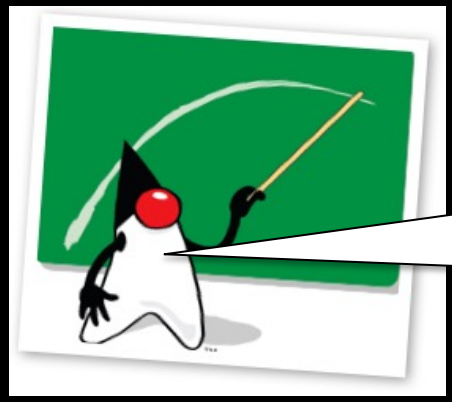


Next, the compiler already knows that **IntConsumer**'s **accept** method takes a parameter of type **int**, so we don't have to say so explicitly: let's remove the type info.

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                      (int n) ->  { System.out.println(n);     }   ,
    |                                      (int n) ->    System.out.println(n);          ,
    |                                          (n) ->    System.out.println(n);          ,
```

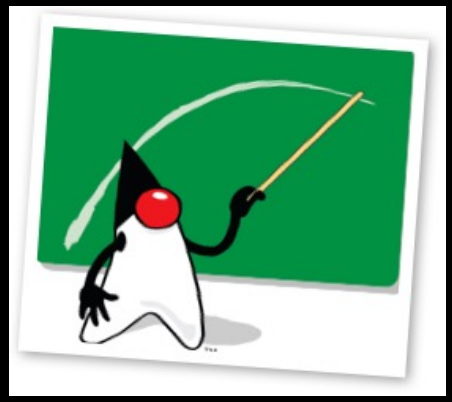


```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                      (int n) ->  { System.out.println(n);     }   ,
    |                                      (int n) ->    System.out.println(n);          ,
    |                                          (n) ->    System.out.println(n);          ,
```



And finally, the accept method has only one parameter, so the compiler doesn't need us to wrap it in parentheses. Let's remove them.

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                      (int n) ->  { System.out.println(n);     }   ,
    |                                      (int n) ->    System.out.println(n);          ,
    |                                          (n) ->    System.out.println(n);          ,
    |                                           n  ->    System.out.println(n);          ,
    | )
java>
```



```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                  (int n) ->  { System.out.println(n);     }   ,
    |                                  (int n) ->    System.out.println(n);        ,
    |                                      (n) ->    System.out.println(n);        ,
    |                                       n  ->    System.out.println(n);        ,
    | )
java>
```

These are all equivalent **lambda expressions**

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                    (int n) ->  { System.out.println(n);     }   ,
    |                                    (int n) ->    System.out.println(n);        ,
    |                                        (n) ->    System.out.println(n);        ,
    |                                         n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
```

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                    (int n) ->  { System.out.println(n);     }   ,
    |                                    (int n) ->    System.out.println(n);        ,
    |                                        (n) ->    System.out.println(n);        ,
    |                                         n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
999
999
999
999
999
999
java>
```

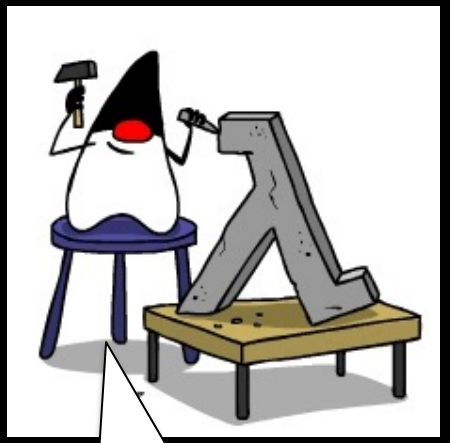
```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                  (int n) ->  { System.out.println(n);     }   ,
    |                                  (int n) ->    System.out.println(n);        ,
    |                                      (n) ->    System.out.println(n);        ,
    |                                       n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
999
999
999
999
999
999
java>
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
```

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                  (int n) ->  { System.out.println(n);     }   ,
    |                                  (int n) ->    System.out.println(n);        ,
    |                                      (n) ->    System.out.println(n);        ,
    |                                       n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
999
999
999
999
999
999
java>
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
1
2
3
java>
```

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                  (int n) ->  { System.out.println(n);     }   ,
    |                                  (int n) ->    System.out.println(n);        ,
    |                                      (n) ->    System.out.println(n);        ,
    |                                       n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
999
999
999
999
999
999
java>
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
1
2
3
java>
```

Instead of passing in an anonymous implementation of **IntConsumer**

```
java>
java>List<IntConsumer> consumers = Arrays.asList(
    | new IntConsumer() { public void accept(int value) { System.out.println(value); } } ,
    | new IntConsumer() { public void accept(int n)     { System.out.println(n);     } } ,
    |                                  (int n) ->  { System.out.println(n);     }   ,
    |                                  (int n) ->    System.out.println(n);        ,
    |                                      (n) ->    System.out.println(n);        ,
    |                                       n  ->    System.out.println(n);        ,
    | )
java>
java> for(IntConsumer consumer : consumers) { consumer.accept(999); }
999
999
999
999
999
999
java>
java> IntStream.of(1, 2, 3).forEach( new IntConsumer() {
    | public void accept(int value) { System.out.println(value); }
    | })
1
2
3
java>
java> IntStream.of(1, 2, 3).forEach( n -> System.out.println(n) )
1
2
3
java>
```



we can pass in a **lambda expression**

Why is that?

Why is it that in Java 8 we can replace this

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

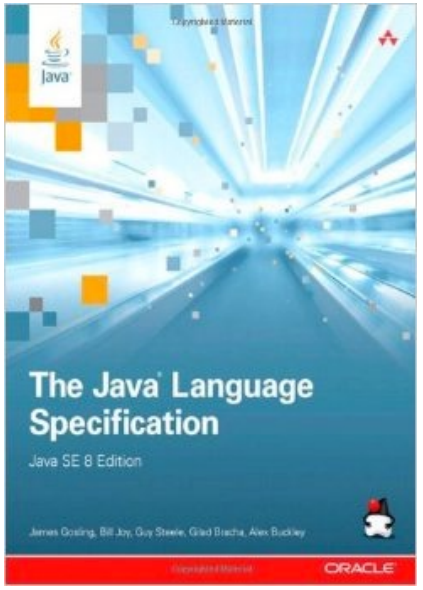
with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

java.lang.stream.**IntConsumer** is a **Functional Interface** (new in Java 8)

A functional interface is an interface that has just one abstract method (aside from the methods of Object), and thus represents a single function contract…

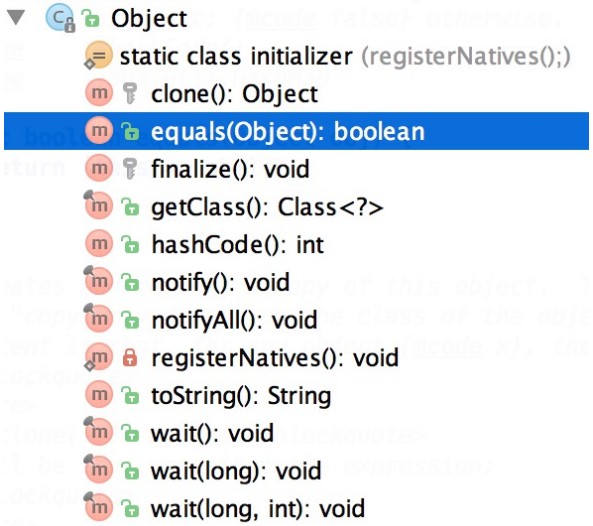


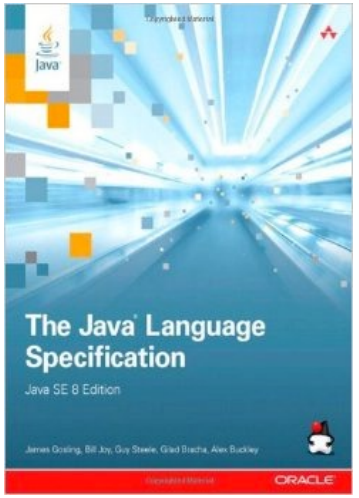
Java SE 8 Edition

A simple example of a functional interface is:

```
interface Runnable {
    void run();
}
```

SAM (**S**ingle **A**bstract **M**ethod)






```
interface NonFunc {
    boolean equals(Object obj);
}
```

Not functional because it declares nothing which is not already a member of Object

```
interface Func extends NonFunc {
    int compare(String o1, String o2);
}
```

Functional because it declares an abstract method which is not a member of Object

SAM (**S**ingle **A**bstract **M**ethod)

```
interface Comparator<T> {
    boolean equals(Object obj);
    int compare(T o1, T o2);
}
```

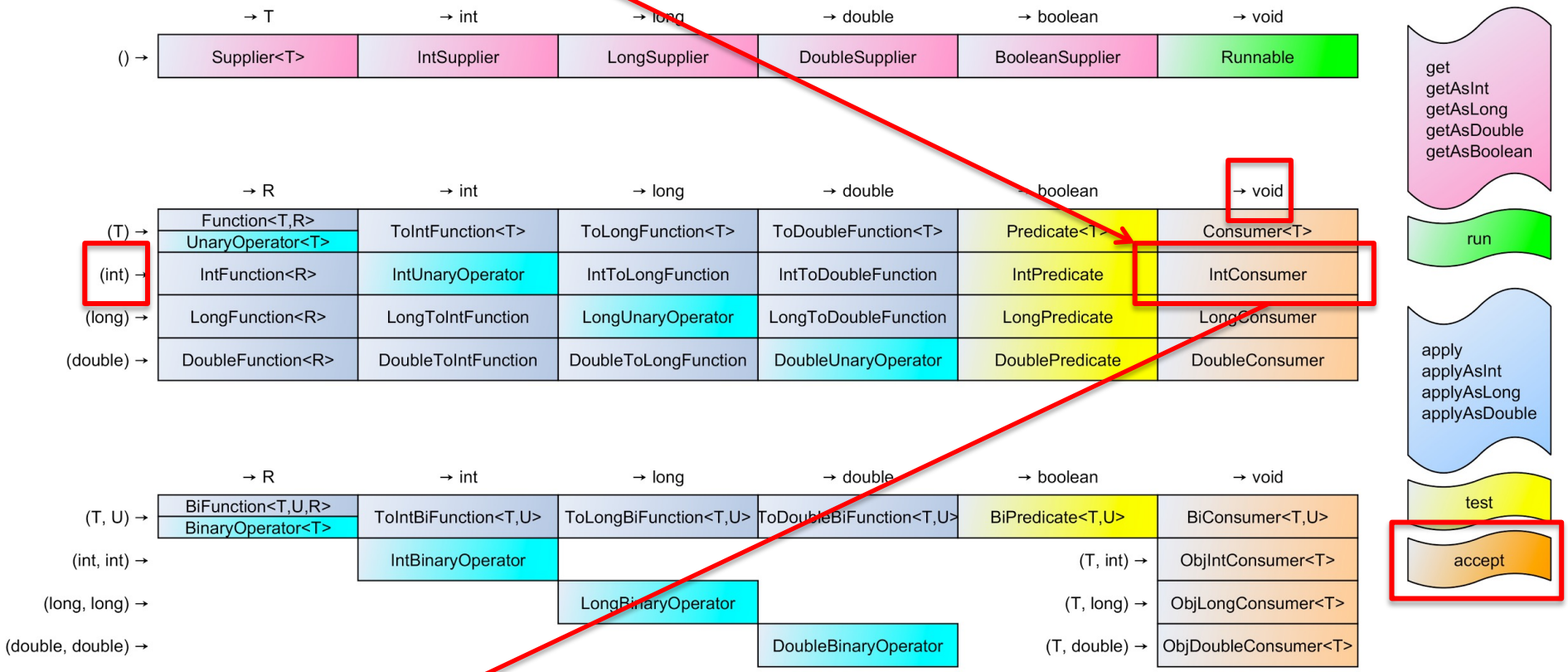
Functional because it has one abstract non-Object method

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

java.lang.stream.**IntConsumer** is a **predefined Functional Interface**

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

Java 8 Predefined Functional Interfaces



Java 8 Functional Interface Naming Guide
by Esko Luontola, www.orfjackal.net

accept

`void accept(int value)`

Performs this operation on the given argument.

For each value **n** in the stream, the **forEach** method calls **accept**(**n**)

Back to the question…

Why is it that in Java 8 we can replace this

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

`n -> System.out.println(n)`

is a Java **lambda expression**

Lambda expressions are named after the **Lambda Calculus**

Lambda because greek letter **λ** is used in **lambda expressions**

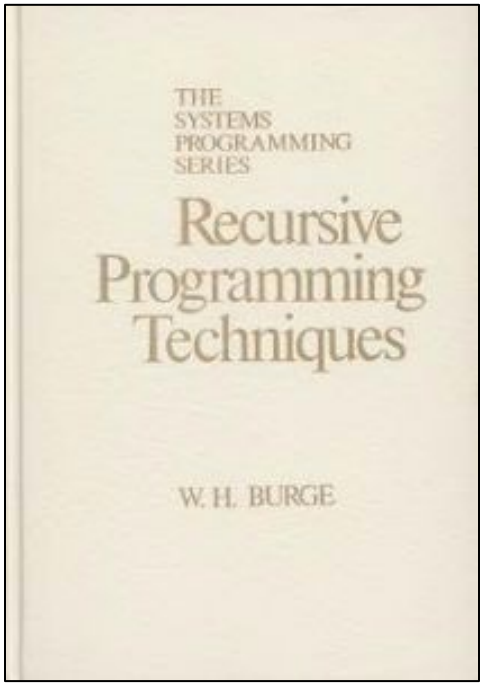
Greek letter lambda

There is no significance to the letter

The **λ calculus** was introduced by American Mathematician Alonzo Church in the 1930s

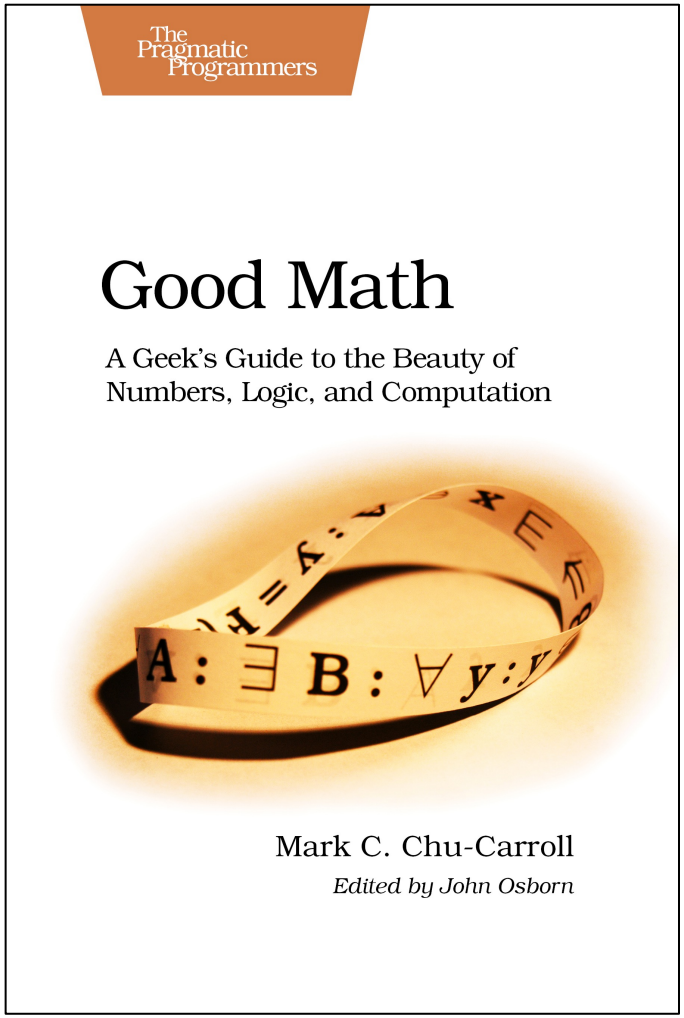
Quick introduction to the **λ calculus**

Material from the following, but mainly from 'Good Math':



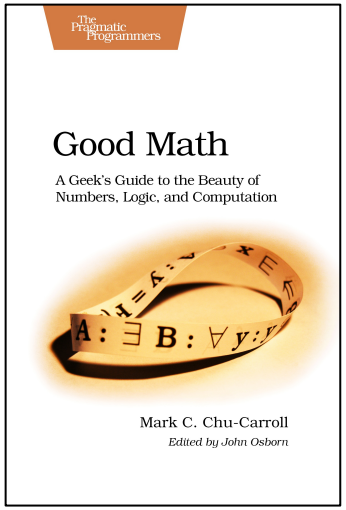
1975

1988



2013

The **λ calculus** is probably the most widely used theoretical tool in **computer science**



In the field of **programming languages**, we use it to

- understand or prove facts about computation
- describe how programming languages work

Functional programming languages like

Haskell

Scala

are so strongly based in **λ calculus** that they're just **alternative syntaxes for pure λ calculus.**

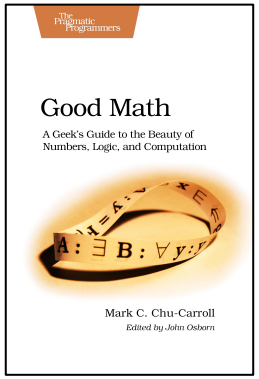
But the influence of **λ calculus** isn't limited to relatively esoteric functional languages

python

Ruby

have strong **λ-calculus** influences

What makes the **λ calculus** so great?



Simple: only three constructs and two evaluation rules

$$T ::= v \mid \lambda v.T \mid T\ T$$
$$v ::= v_1 \mid v_2 \ldots$$

α conversion
β reduction

Turing complete: if a function can be Computed by any possible computing device, then it can be written in **λ calculus**

Alan Turing

Easy to read and write: syntax looks like a programming language

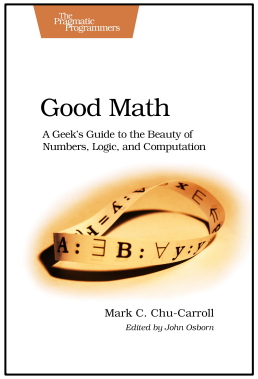
easy!

Strong semantics: based on a logical structure with solid formal model, so very **easy to reason about how it behaves**

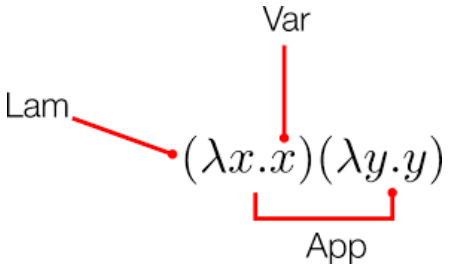
Flexible: simple structure, so easy to create variants for exploring various alternative ways of structuring computations or semantics

The **λ calculus** is based on the concept of functions

$$f(x)$$



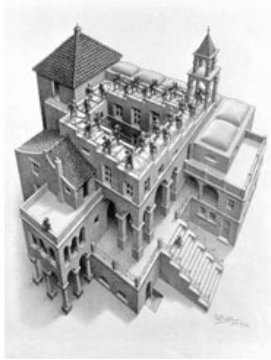
The basic expression in **λ calculus** is the Definition of a function in a special form, called a **λ expression**



In pure **λ calculus**, **everything is a function**; **the only thing you can do is define and apply functions, so there are no values except for functions**

λλλλλλλλλ
λλλλλλλλλ
λλλλλλλλλ
λλλλλλλλλ

As peculiar as that may sound, **it's not a restriction at all: we can use λ calculus functions to create any data structure we want**



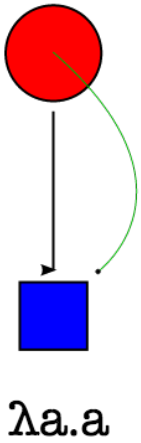
Syntax of Lambda Expressions

<λ-expression> ::= <variable>|<abstraction>|<application>

<abstraction> ::= λ<variable>.<λ-expression>

<application> ::= <λ-expression><λ-expression>

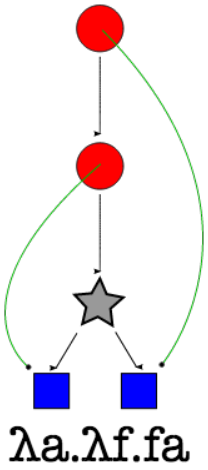
<λ-expression>
<abstraction>
λ**<variable>**.<λ-expression>
λ**x**.**<λ-expression>**
λx.**<variable>**
λx.**x**



λx.x aka **I**, the identity combinator

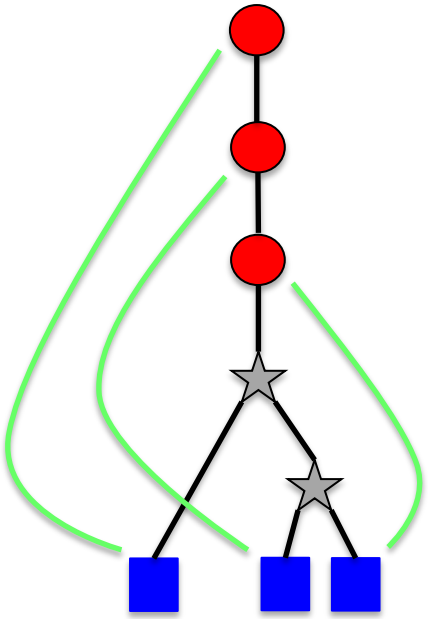
λx.x and λa.a are the same thing

<λ-expression>
<abstraction>
λ**<variable>**.<λ-expression>
λ**a**.**<λ-expression>**
λa.**<abstraction>**
λa.λ**<variable>**.<λ-expression>
λa.λ**f**.**<λ-expression>**
λa.λ**f**.**<application>**
λa.λf.**<λ-expression>**<λ-expression>
λa.λf.**<variable>**<λ-expression>
λa.λf.**f<λ-expression>**
λa.λf.**f<variable>**
λa.λf.f**a**



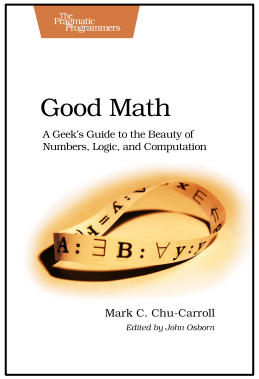
λa.λf.fa takes a parameter and a function and applies the function to the parameter

<abstraction>
λ<variable>.<λ-expression>
λf.<λ-expression>
λf.<abstraction>
λf.λ<variable>.<λ-expression>
λf.λg.<λ-expression>
λf.λg.<abstraction>
λf.λg.λ<variable>.<λ-expression>
λf.λg.λx.<λ-expression>
λf.λg.λx.<application>
λf.λg.λx.<λ-expression><λ-expression>
λf.λg.λx.<variable><λ-expression>
λf.λg.λx.f<λ-expression>
λf.λg.λx.f<application>
λf.λg.λx.f<λ-expression><λ-expression>
λf.λg.λx.f<variable><λ-expression>
λf.λg.λx.f(g<λ-expression>)
λf.λg.λx.f(g<variable>)
λf.λg.λx.f(gx)



λf.λg.λx.f(g x)

λf.λg.λx.f(g x) – The **compose** function: takes a function **f** and a function **g** and returns a function that takes a parameter **x**, applies **g** to it, and then applies **f** to the result



There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

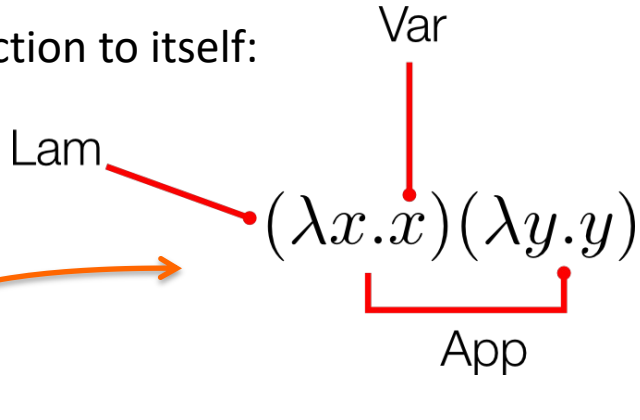
only useful when evaluating complex expressions,
to avoid variable name collisions

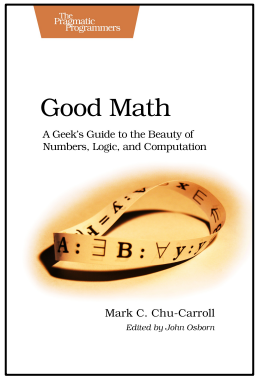
β reduction is how functions are applied: if you have a function application, you can apply it by

1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β--->





There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

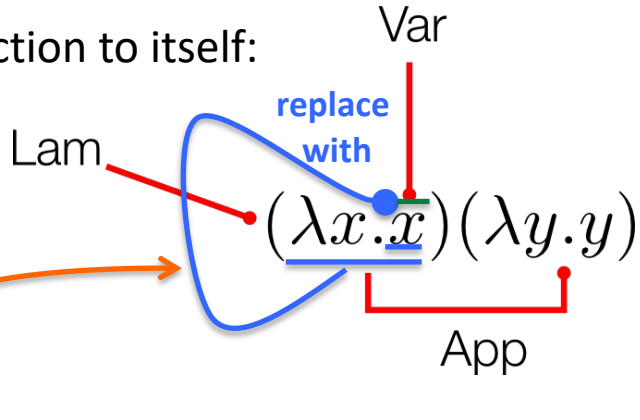
only useful when evaluating complex expressions,
to avoid variable name collisions

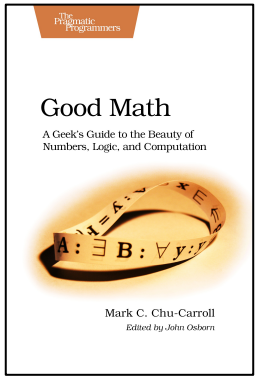
β reduction is how functions are applied: if you have a function application, you can apply it by

1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β--->





There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

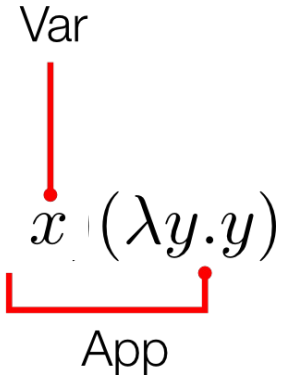
only useful when evaluating complex expressions,
to avoid variable name collisions

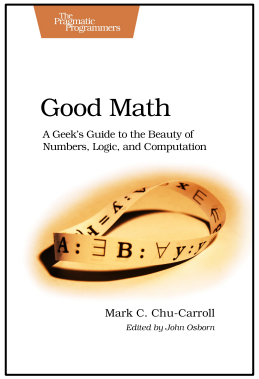
β reduction is how functions are applied: if you have a function application, you can apply it by

1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β--->





There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

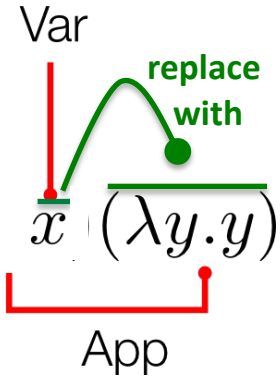
only useful when evaluating complex expressions,
to avoid variable name collisions

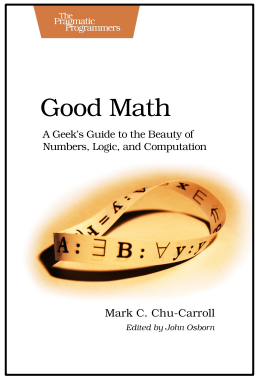
β reduction is how functions are applied: if you have a function application, you can apply it by

1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β--->





There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

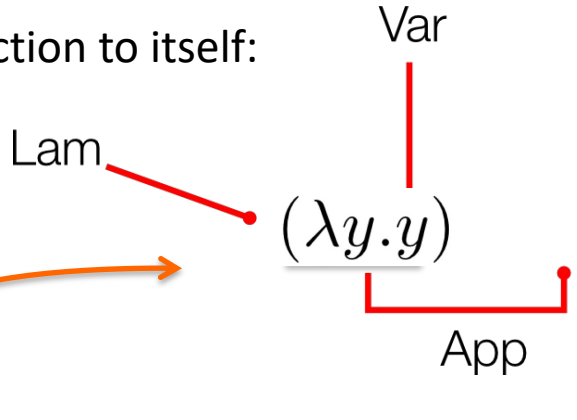
only useful when evaluating complex expressions,
to avoid variable name collisions

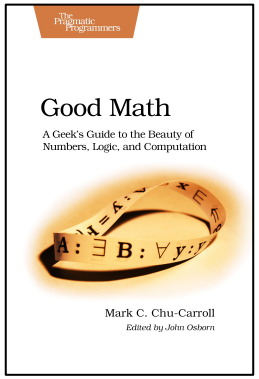
β reduction is how functions are applied: if you have a function application, you can apply it by

1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β--->





There are **only two real rules for evaluating expressions** in **λ calculus**:
α conversion and **β reduction**

α conversion is just renaming of a variable in an expression

λx.x ----α---> **λy.y**

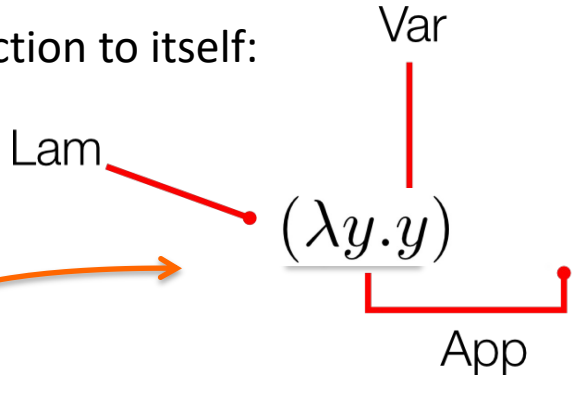
only useful when evaluating complex expressions,
to avoid variable name collisions

β reduction is how functions are applied: if you have a function application, you can apply it by

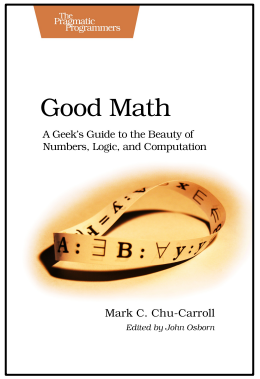
1. First **replacing** the function with its body
2. and then taking the argument expression and **replacing** all occurrences of the parameter in the body with the argument expression

As an example of **β reduction**, let's apply the identity function to itself:

I I = **λx.x(λx.x)** ----α---> **λx.x(λy.y)** ----β---> **λy.y** = **I**



Question: If all we can do in a **λ calculus** computation is just **β reduction** and **α renaming**, then why is the following true?

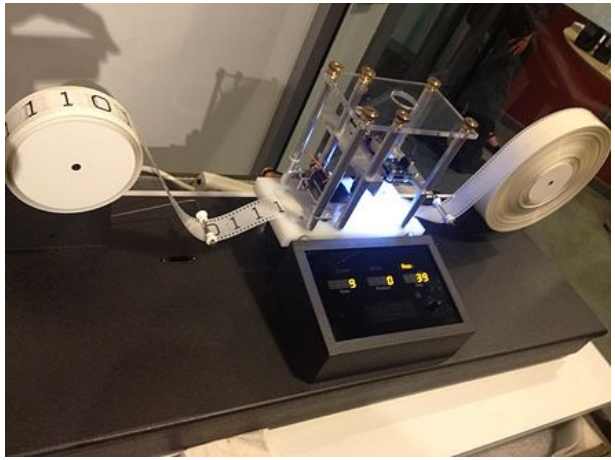


if a function can be computed by any possible computing device, then it can be written in λ calculus

Answer:

The **λ calculus** **is** **Turing complete**, i.e. **it can do the same computations as a Turing machine**

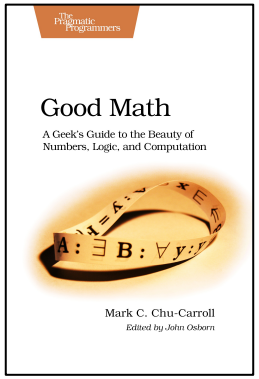
The **Turing machine** is an example of the **maximum capability of any mechanical computer**



The capability of a **Turing machine** can be matched, but never exceeded.

If a computing device can do the same computations as a Turing machine, then it can do everything that any computer can do

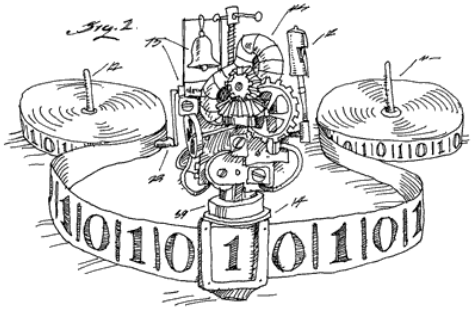
But **why is the λ calculus Turing complete**?



To answer that, let's look at **what a machine needs to be Turing complete**

How hard is it to make **machines that can perform any possible computation**?

The answer is, surprisingly easy



The truth is that:

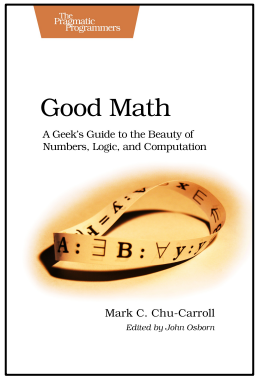
the basics of computation are so simple that you have to work hard to build a computing system that isn't Turing complete

What a machine needs to be **Turing complete**:

- ☐ Unlimited Storage
- ☐ Arithmetic
- ☐ Conditional Execution
- ☐ Repetition

Let's look at each of these in turn and show that the λ calculus is **Turing complete**

Unlimited Storage



To be **Turing complete**, you can't have any fixed limits on the amount of storage you can access.

It doesn't matter what kind of storage it is, as long as it's unlimited

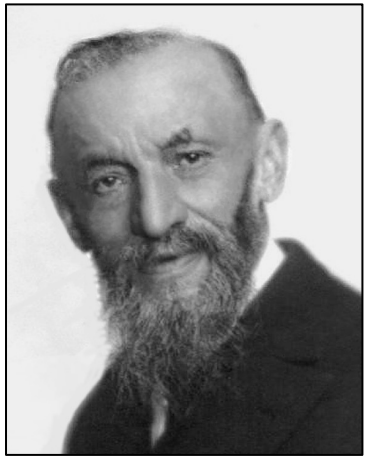
In the **λ calculus**, the storage part is easy:

- we can store arbitrarily complicated values in a variable
- we can generate as many functions as we need without bound

So unbounded storage is pretty obvious

- [x] Unlimited Storage
- [ ] Arithmetic
- [ ] Conditional Execution
- [ ] Repetition

Arithmetic



Your machine must at least be able to perform the basic operations defined by **Peano arithmetic**

Peano Arithmetic (Giuseppe Peano 1858 – 1932)

Axioms

1) $0 \in N$ — There exists a natural number zero

2) $\forall n \in N \; n' \in N$ — For each natural number n, there exists a natural number that is the successor of n, denoted by n'

3) $\forall n \in N \; n' \neq 0$ — Zero is not the successor of any natural number

4) $\forall n,m \in N \; n' = m' \rightarrow n = m$ — If the successor of two natural numbers is the same, then the two original numbers are the same.

5) $P(0), \forall n \in N \; (P(n) \rightarrow P(n')) \vdash \forall n \in N \; P(n)$ — The principle of mathematical induction

Operations

Addition

$\forall n \in N \quad n + 0 = n$

$\forall n,m \in N \quad n + m' = (n + m)'$

Multiplication

$\forall n \in N \quad n * 0 = 0$

$\forall n,m \in N \quad n * m' = n * m + n$

Peano Arithmetic in the Lambda Calculus

$0 = \lambda f.\lambda x.x$
$1 = \lambda f.\lambda x.fx$
$2 = \lambda f.\lambda x.f(fx)$
$3 = \lambda f.\lambda x.f(f(fx))$

Church Numerals

Named after
Alonzo Church

succ $= \lambda n.\lambda f.\lambda x.f(nfx)$

- [x] Unlimited Storage
- [x] Arithmetic
- [ ] Conditional Execution
- [ ] Repetition

$+ = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$
$* = \lambda m.\lambda n.\lambda f.m(nf)$

We can now start writing more interesting λ-expressions, e.g.

$(\lambda x.+xx)2$ ----β---> 4 - the **double** function

$(\lambda x.*xx)3$ ----β---> 9 - the **square** function

Conditional Execution

To do general computation, you need to have some way of **making choices**

You need to have the **ability to select different behaviors** based on values that were either computed or were part of the input to your program

Conditional Execution in the Lambda Calculus

```
true  = λx.λy.x      true 3 4  ----β---> 3
false = λx.λy.y      false 3 4 ----β---> 4

if C then P else Q = CPQ   →   λc.λp.λq.cpq

and = λx.λy.x y false
or  = λx.λy.x true y
not = λx.x false true

isZero = λn.n(true false)true

λx.if (isZero x) then 10 else + x x
```

- ☑ Unlimited Storage
- ☑ Arithmetic
- ☑ Conditional Execution
- ☐ Repetition

Repetition

You need loops, recursion, or some other repetition for iterating or repeating pieces of a computation

And they **need to be able to work together with** the mechanism for **conditional execution** to allow conditional repetition

Since everything in the lambda calculus is a function, let's look at recursive functions.

Sample recursive function: **Factorial**.

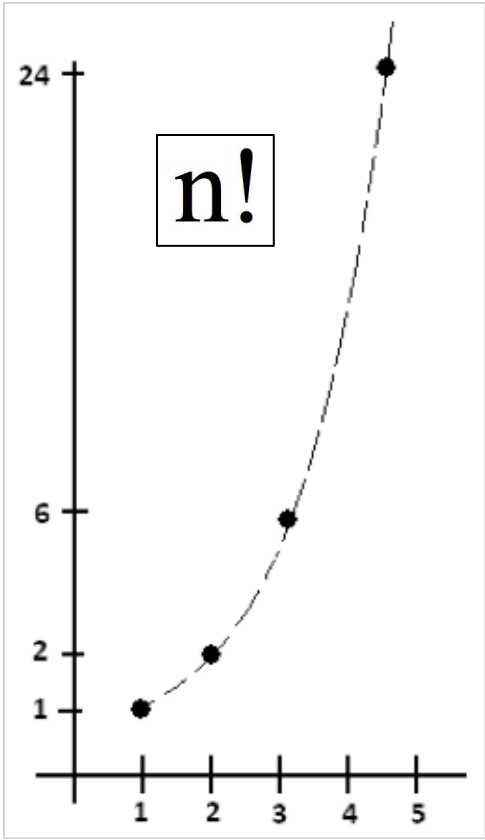
!

Factorial

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n \cdot (n-1)! & \text{if } n>0 \end{cases}$$

$$n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$$

```
0! = 1
1! = 1
2! = 2 x 1 = 2
3! = 3 x 2 x 1 = 6
4! = 4 x 3 x 2 x 1 = 24
5! = 5 x 4 x 3 x 2 x 1 = 120
...
```



Repetition

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n \cdot (n-1)! & \text{if } n>0 \end{cases}$$

```
factorial(n): if n=0 then 1 else n * factorial(n - 1)
```

In the lambda calculus:

```
pred = λn.(((n)λp.λz.((z)(succ)(p)(true)(p)true)λz.((z)0)0)false
```

```
fact = λn.if (isZero n) then 1 else * n (fact (pred n))
```

This is circular. What do we do next?

Let's abstract out the recursive call:

```
fact = (λf.λn.if (isZero n) then 1 else * n (f(pred n)))fact
```

Let's name the λ expression **metaFact**, since it uses **fact**, which it is passed as an argument

```
fact = metaFact fact
```

To find **fact**, we must solve this equation

Consider these fixpoint equations:

```
n = double n   n=0, since 0 = double 0
n = square n   n=1, since 1 = square 1
e = Ie         e=anything, since I just returns its param
```

The solutions to these equations are **fixed-points** of functions **double**, **square** and **I**.

The equations are all of the following form:

```
x = fx
```

Contrast them with

```
fact = metaFact fact
```

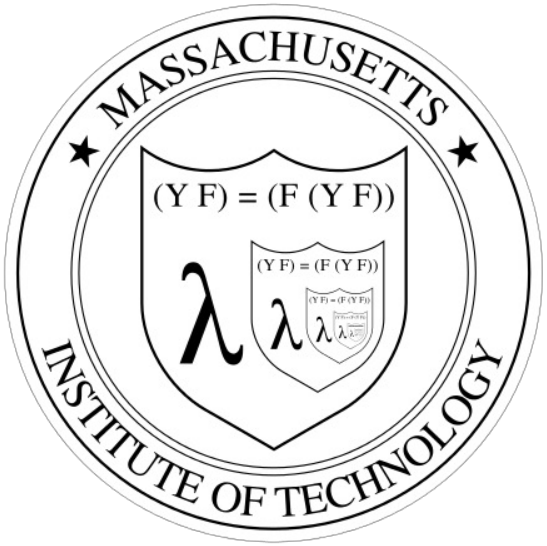
which is of the form

```
f = Ef
```

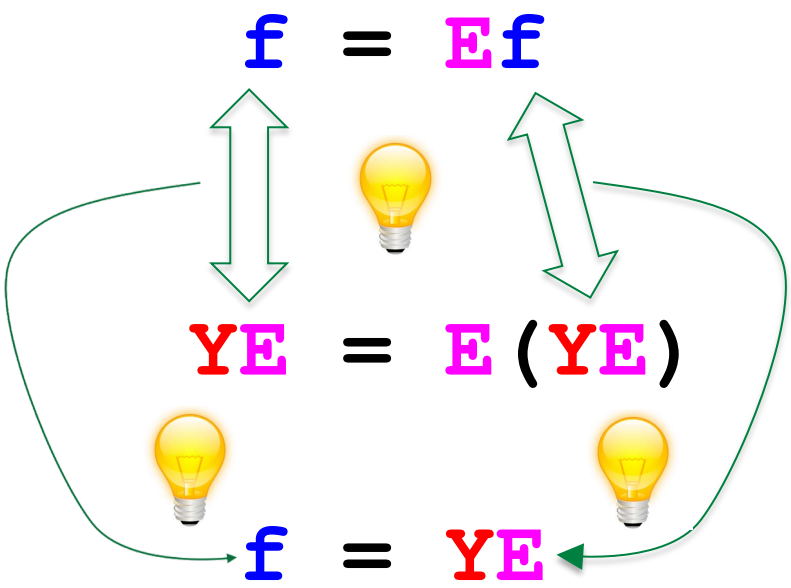
Every recursive definition can be brought to this form

The solution to this equation is **f**, the **fixed point** of higher order function **E**.

Conveniently, in the λ calculus there is a "**special, almost magical function**" that makes recursion possible: **Y**, the **fixed-point** combinator.



$$YE \xrightarrow{\beta^*} E(YE) \xrightarrow{\beta^*} E(E(YE)) \xrightarrow{\beta^*} \ldots \xrightarrow{\beta^*} E(E(E\ldots(YE)\ldots))$$

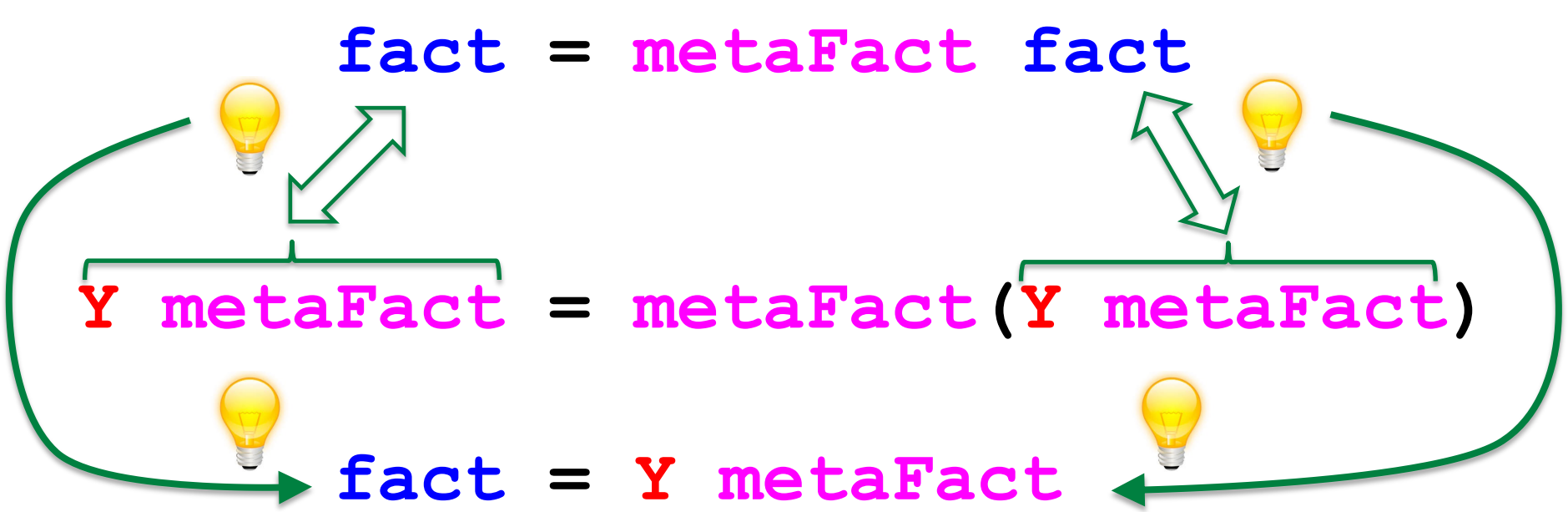


which is **no longer recursive !!!**

The equation we have to solve.
What is **f**, the **fixed point** of HoF **E**?

True for every λ-expression **E**, by definition of **Y**

The solution: applying **fixed point combinator Y** to **E** gives us **f**, the fixed point of **E**



which is **no longer recursive !!!**

Expanding **metafact**:

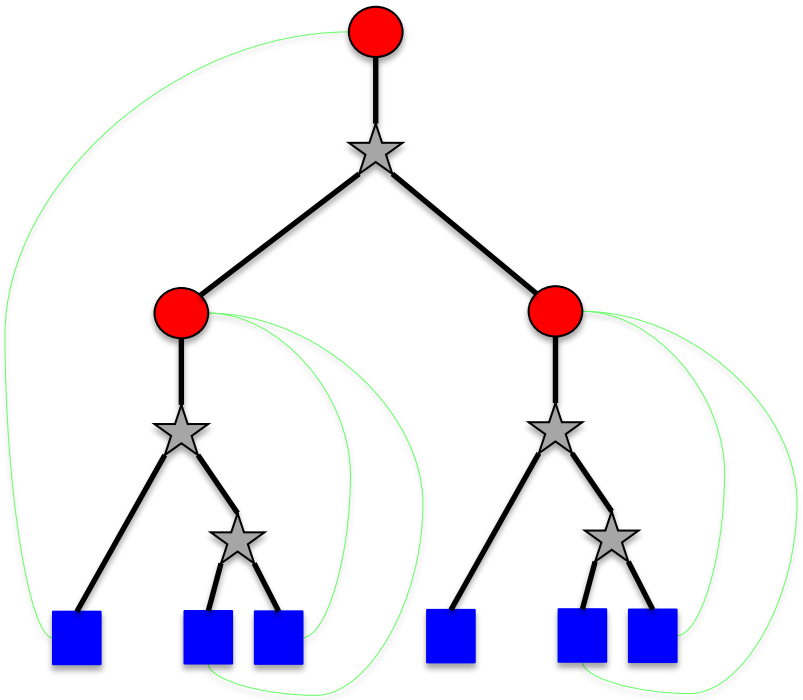
```
fact = Y(λf.λn.if (isZero n) then 1 else * n (f(pred n)))
```

Again: **no longer recursive !!!**

- ☑ Unlimited Storage
- ☑ Arithmetic
- ☑ Conditional Execution
- ☑ Repetition

The reason for calling the fixed-point combinator **Y** is because it is shaped like a Y

$\lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$



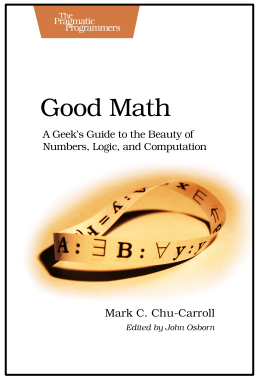
Now let's see why $Yf = f(Yf)$:

$Y = \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$

$Yf = (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))f \quad \text{----}\beta\text{--->} \quad (\lambda x.f(xx))(\lambda x.f(xx))$

$\text{----}\beta\text{--->} \quad f((\lambda x.f(xx))(\lambda x.f(xx)))$

$= f(Yf)$ **Q.E.D.**



So, the β rule is **all that's needed** to make the **λ calculus** capable of performing **any computation that can be done by a machine**

All computation is really just β reduction, with **α renaming** to prevent name collisions

The Turing machine is one of the simpler ways of doing computation

The **λ calculus** is **a lot simpler than** the state-plus-tape notion of the **Turing machine**

The λ calculus is the simplest formal system of computation

There is Duke, looking very smug

Congratulations Duke: you have just caught up with where Alonzo Church was in the 1930s!

Computer Scientist Philip Wadler

Alonzo Church

Let’s see some ways in which lambda expressions differ in **Java** and in **λ calculus**

λx . x

x -> x

In **Java**, **λ** and **.** are replaced by **->**

`(x, y) -> x + y`

`λxy.+xy`

`λx.λy.+xy`

This **Java** function takes two parameters

Whereas in the **λ calculus** functions can have only one parameter

Does this mean the **λ calculus** is underpowered?

No, because a **function** that takes **n** parameters can be represented by **n functions** that take **1** parameter

This is called **currying**

after logician **Haskell Curry**

So we can have the convenience of multi-parameter functions in the **λ calculus** too

When can write **λxy.+xy**, which is just **syntactic sugar** for the curried version: **λx.λy.+xy**

```
(x, y) -> x+y        λxy.**xy
```

```
λxy.+xy
```

In the **Java** function the plus is in **infix** position ,
whereas in the lambda calculus it is in **prefix** position

We can have infix notation in the **λ calculus** too if we like ,

it can just be treated as **syntactic sugar** for the prefix notation

```
(int x, int y) -> x + y
```

λ x:α.λy:α.x+y

In **Java**, function parameters always have a type, although sometimes it can be left implicit

The **return type** is always **implicit**. In this case it is **int**

We have looked at the **untyped λ calculus**, in which function application has no restrictions

There are typed varieties of **λ calculus** that restrict function application, so that a **function can only be applied to actual parameters whose type matches that of the function's formal parameters**

λ x: **α** . <λ-expression> λ x. <λ-expression>:**α → α**

λ x: **N** . x + 3 (λ x . x + 3): **N → N**

(λ x: **N** . x + 3)2 **Valid lambda expression**: a function expecting an N is applied to an N

(λ x: **N** . x + 3)**true** **Invalid**: a boolean value is not a natural number

Syntax of Lambda Expressions in λ-calculus

<λ-expression> ::= **<variable>|<application>|<abstraction>**

<application> ::= <λ-expression><λ-expression>

<abstraction> ::= **λ**<variable>.<λ-expression>

body = <variable>	`λx.x`	`x -> x`
body = <abstraction>	`λx.λy.+xy`	`x -> (y -> x+y)`

In the **λ-calculus** the application of a function **f** to some argument **v** is expressed simply by juxtaposing the two: **fv**. **In Java 8 however, application of f to v is expressed as f.apply(v)**

	`square = λn.*nn`	`square = n -> n*n`
body = <application>	`λx.square x`	`x -> square.apply(x)`

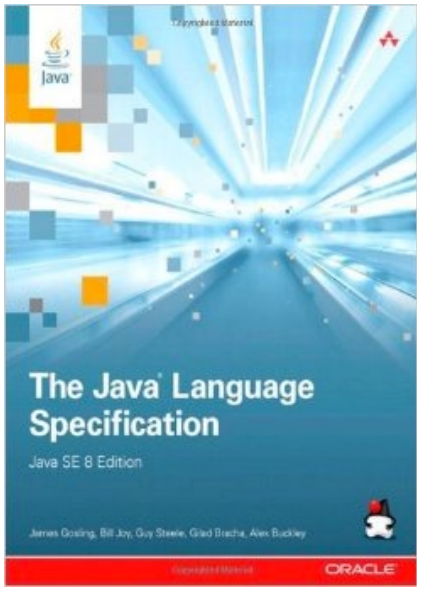
We'll soon see why **apply** is needed in Java 8.

"A lambda expression is **like a method**: it provides **a list of formal parameters and a body** - an expression or block - expressed in terms of those parameters"

```
(x,y) -> x+y
```

lambda-expression **body**: an **expression** or **block**

```
flag ->
{ if (flag) return 12;
  else {
    int result = 15;
    for (int i = 1; i < 10;
i++)
      result *= i;
    return result;
  }
}
```



Java SE 8 Edition

Can't do that in the **λ calculus**

```
(n) -> { System.out.println(n); return n; }
```

In **Java**, functions can have side effects. Can't do that in the **λ calculus**

```
n -> System.out.println(n)
```

In **Java**, functions don't have to have a value: their return type can be void

Here is an interesting example: `n -> {}`

```
() -> System.out.println("Hello World!")
```

In **Java**, functions don't have to have parameters

Here is an interesting example: `() -> {}`

Back to the question…

Why is it that in Java 8 we can replace this

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

Back to the question…

Why is it that in Java 8 we can replace this

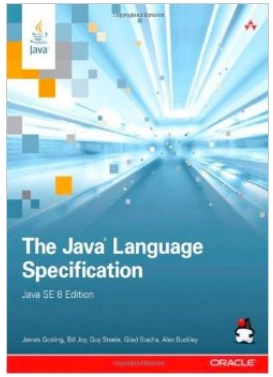
```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

Because:

"**Evaluation of a lambda expression** produces **an instance of a functional interface**"



Back to the question…

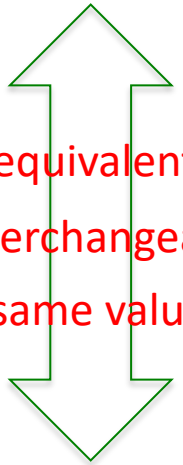
Why is it that in Java 8 we can replace this

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

Anonymous Instance of **Functional Interface**



equivalent

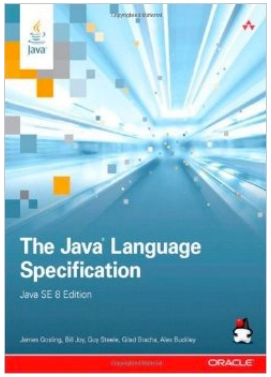
interchangeable

same value

Lambda Expression

Because:

"**Evaluation of a lambda expression** produces **an instance of a functional interface**"



Back to the question…

Why is it that in Java 8 we can replace this

```
IntStream.of(1, 2, 3).forEach(
    new IntConsumer() {
        public void accept(int value) {
            System.out.println(value);
        }
    }
)
```

Anonymous Instance of
Functional Interface

equivalent
interchangeable
same value

Lambda Expression

with this?

```
IntStream.of(1, 2, 3).forEach(
    n -> System.out.println(n)
)
```

This is possible because **a Functional Interface has just one method**
So the **compiler can check that**

1) the lambda expression's parameters, match those of the method's signature
2) the return type of the lambda expression's body matches the return type of the signature

one int parameter ☑

accept

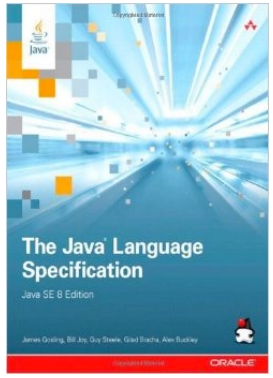
`void accept(int value)`

Performs this operation on the given argument.

void return type ☑

Because:

"**Evaluation of a lambda expression** produces **an instance of a functional interface**"



Functional Interface (SAM)

```
public interface I
{
    public R m(A a, B b);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new I()
{
    public R m(A a, B b)
    {
        <body>
    }
}
```

<body> is 1 or more statements,
and the last one returns an R

Functional Interface (SAM)

```
public interface I
{
    public R m(A a, B b);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new I()
{
    public R m(A a, B b)
    {
        <body>
    }
}
```

<body> is 1 or more statements,
and the last one returns an R

Replace with

Lambda Expression

```
(a, b) -> { <body> }
```

Functional Interface (SAM)

```
public interface I
{
    public R m(A a, B b);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new I()
{
    public R m(A a, B b)
    {
        <body>
    }
}
```

`<body>` is 1 or more statements, and the last one returns an R

Replace with

Lambda Expression

```
(a, b) -> { <body> }
```

(Syntactic Sugar)

Functional Interface (SAM)

```
public interface I
{
    public R m(A a, B b);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new I()
{
    public R m(A a, B b)
    {
        <body>
    }
}
```

<body> is 1 or more statements,
and the last one returns an R

Equivalent

Lambda Expression

```
(a, b) -> { <body> }
```

(Syntactic Sugar)

Functional Interface (SAM)

```
public interface I
{
    public R m(A a, B b);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new I()
{
    public R m(A a, B b)
    {
        <body>
    }
}
```

`<body>` is 1 or more statements, and the last one returns an R

Equivalent

Lambda Expression

```
(a, b) -> { <body> }
```

(Syntactic Sugar)

ADVISORY

take claims of lambda expressions being syntactic sugar for anonymous inner classes with a pinch of salt

```
T p(I i){ … i.m(…,…) …}
```

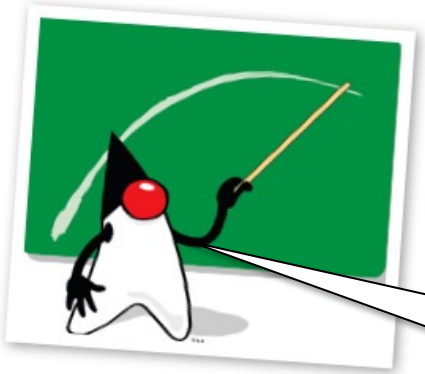
```
p( new I() { public R m(A a, B b) { <body> }  } )
```

simplify call site

```
p( (a, b) -> { <body> } )
```

```
…
i.m(…,…)
…
```

```
I i = new I() { public R m(A a, B b) { <body> }  };
```



Simplify declaration

```
I i = (a, b) -> { <body> };
```

Why would you want to do that?
Name an anonymous function?

```
T p(I i){ … i.m(…,…) …}
```

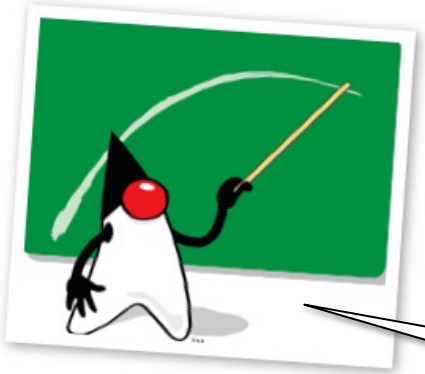
```
p( new I() { public R m(A a, B b) { <body> }  } )
```

simplify call site

```
p( (a, b) -> { <body> } )
```

```
…
i.m(…,…)
…
```

```
I i = new I() { public R m(A a, B b) { <body> }  };
```



Simplify declaration

```
I i = (a, b) -> { <body> };
```

We'll see why later on

Functional Interface (SAM)

```
public interface IntConsumer
{
    public void accept(int value);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new IntConsumer()
{
    public void accept(int value)
    {
        System.out.println(value);
    }
}
```

Functional Interface (SAM)

```
public interface IntConsumer
{
    public void accept(int value);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new IntConsumer()
{
    public void accept(int value)
    {
        System.out.println(value);
    }
}
```

Replace with

Lambda Expression

```
value -> System.out.println(value);
```

Functional Interface (SAM)

```
public interface IntConsumer
{
    public void accept(int value);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new IntConsumer()
{
    public void accept(int value)
    {
        System.out.println(value);
    }
}
```

Replace with

Lambda Expression

```
value -> System.out.println(value);
```

Functional Interface (SAM)

```
public interface IntConsumer
{
    public void accept(int value);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new IntConsumer()
{
    public void accept(int value)
    {
        System.out.println(value);
    }
}
```

Same Value

Lambda Expression

```
value -> System.out.println(value);
```

Functional Interface (SAM)

```
public interface IntConsumer
{
    public void accept(int value);
}
```

→ **SAM** (**S**ingle **A**bstract **M**ethod)

Anonymous Implementation

```
new IntConsumer()
{
    public void accept(int value)
    {
        System.out.println(value);
    }
}
```

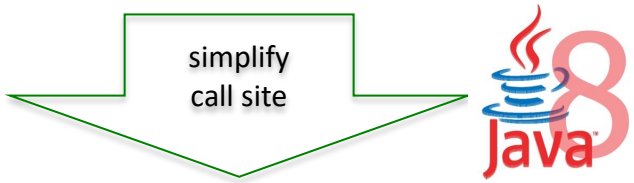
Equivalent

Lambda Expression

```
value -> System.out.println(value);
```

```
void forEach(IntConsumer action){ … action.accept(…) …}
```

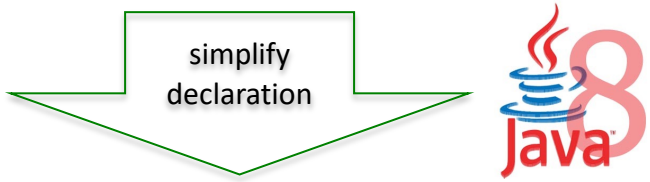
```
forEach( new IntConsumer() {
    public void accept(int value) {
        System.out.println(value); } } )
```




```
forEach( n -> System.out.println(n) )
```

```
…
action.accept(…)
…
```

```
IntConsumer action = new IntConsumer() {
    public void accept(int value) {
        System.out.println(value); }  };
```



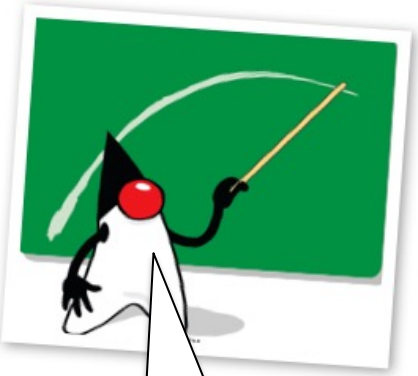

```
IntConsumer action = n -> System.out.println(n);
```

```
void forEach(IntConsumer action){ … action.accept(…) …}
```

```
forEach( new IntConsumer() {
    public void accept(int value) {
        System.out.println(value); } } )
```

simplify call site

```
forEach( n -> System.out.println(n) )
```



Why would you want to do that? Name an anonymous function?

```
…
action.accept(…)
…
```

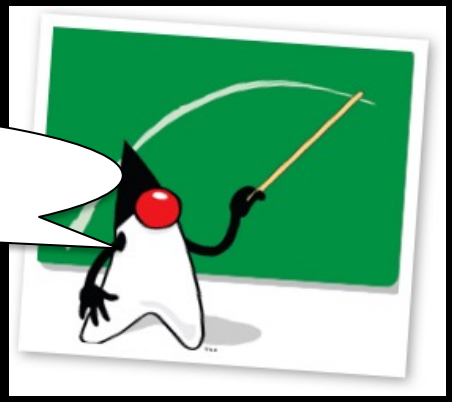
```
IntConsumer action = new IntConsumer() {
    public void accept(int value) {
        System.out.println(value); }  };
```

simplify declaration

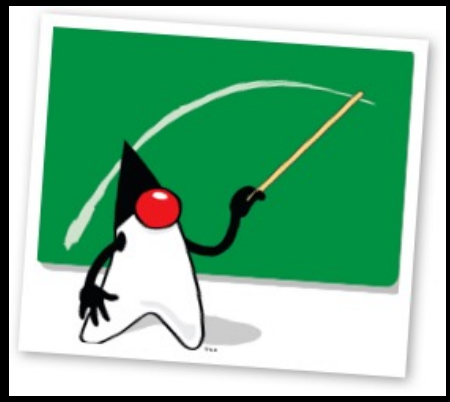
```
IntConsumer action = n -> System.out.println(n);
```

```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
```

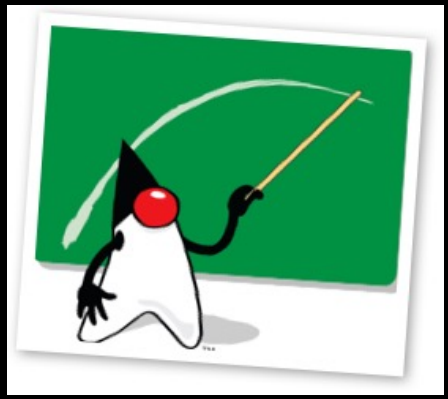
```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
```



```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
```

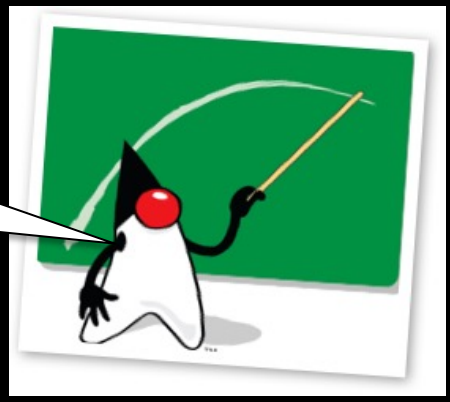


```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```

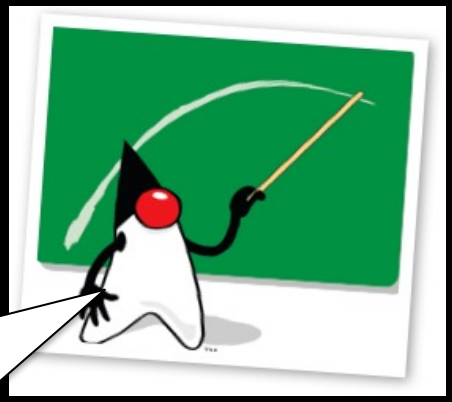


```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```

for readability and reuse

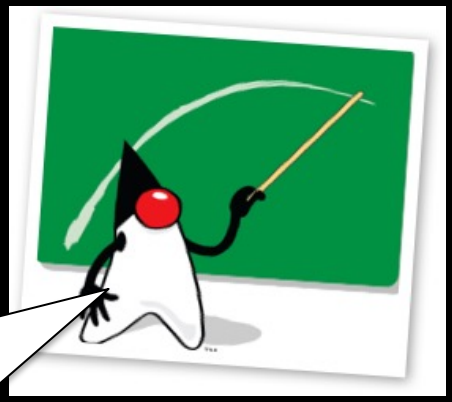


```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```



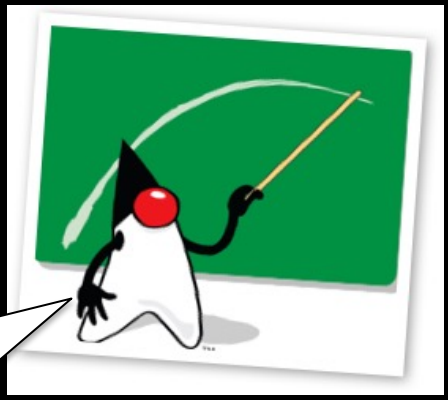
When we first see this, it may look like a method is being passed around in variable **println**, and method **foreach** takes a method as a parameter

```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```



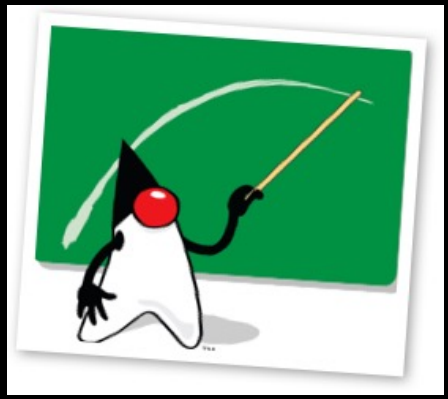
it may kind of **look like 'methods' are now a 'thing'**, that we can pass them around and parametrise our logic with them

```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```

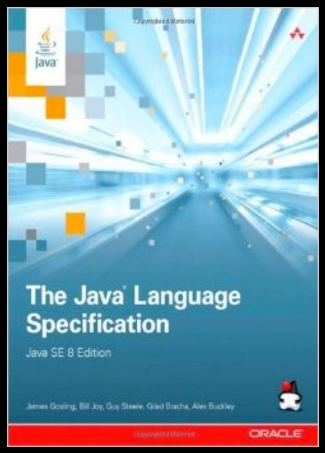


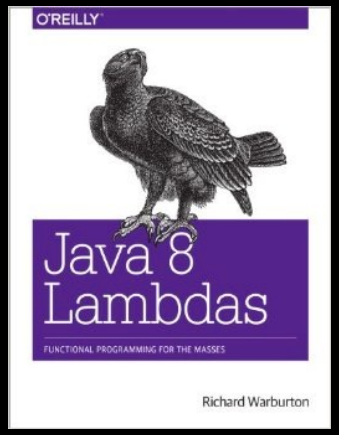
It may look like we can now write **Higher Order** code that takes a method as a parameter, and/or returns a method

```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```



"A lambda expression is like a method: it provides **a list of formal parameters and a body** - an expression or block - expressed in terms of those parameters"

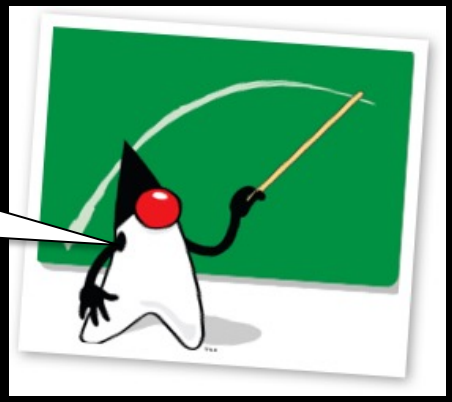




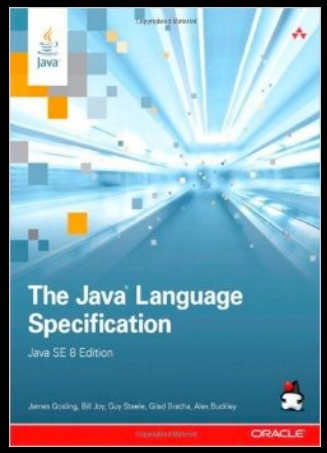
"A lambda expression is a method without a name that is used to pass around behavior as if it were data"

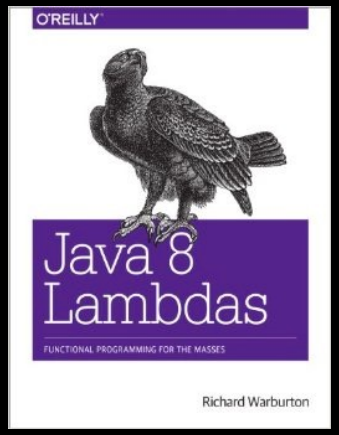
```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```

Statements like these may lead us to think that methods are now **first class citizens**



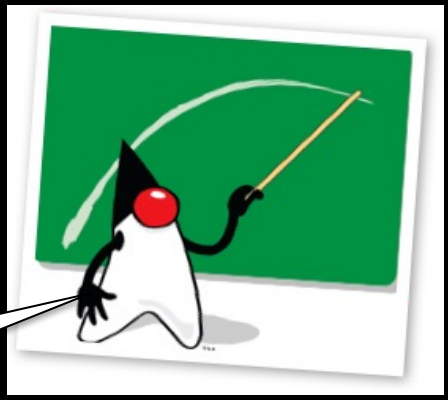
"A lambda expression is like a method: it provides **a list of formal parameters and a body** - an expression or block - expressed in terms of those parameters"





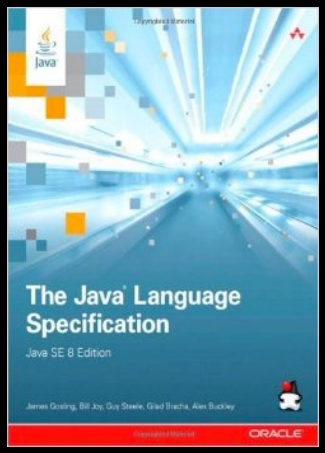
"A lambda expression is a method without a name that is used to pass around behavior as if it were data"

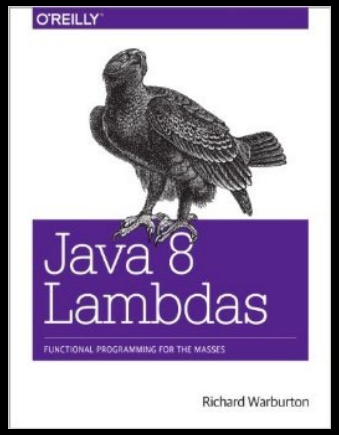
```
java> IntConsumer println = n -> System.out.println(n)
java.util.function.IntConsumer println =
Evaluation$0l26quy1dh8pczmv5j3g$$Lambda$93/1842259508@1cdd5298
java>
java> IntStream.of(1, 2, 3).forEach(println)
1
2
3
java>
java> IntStream.of(4, 5, 6).forEach(println)
4
5
6
java>
```



Let's see another example

"A lambda expression is like a method: it provides **a list of formal parameters and a body** - an expression or block - expressed in terms of those parameters"





"A lambda expression is a method without a name that is used to pass around behavior as if it were data"

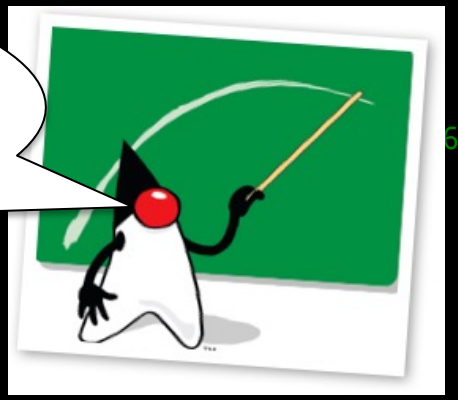
```
java>
java> IntConsumer writeWarningLineNo = n -> System.out.println(n);
java.util.function.IntConsumer writeWarningLineNo =
Evaluation$cb1ez9x8w0oh3urymia5$$Lambda$107/840823054@5db3dd7e
java>
java> IntConsumer writeErrorLineNo   = n -> System.err.println(n);
java.util.function.IntConsumer writeErrorLineNo =
Evaluation$ukw31ztfbqldmcxa50pg$$Lambda$108/1782133702@7d2c1401
java>
java> boolean contextIsError () { return new Random().nextBoolean(); }
Created method boolean contextIsError()
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
```

```
java>
java> IntConsumer writeWarningLineNo = n -> System.out.println(n);
java.util.function.IntConsumer writeWarningLineNo =
Evaluation$cb1ez9x8w0oh3urymia5$$Lambda$107/840823054@5db3dd7e
java>
java> IntConsumer writeErrorLineNo   = n -> System.err.println(n);
java.util.function.IntConsumer writeErrorLineNo =
Evaluation$ukw31ztfbqldmcxa50pg$$Lambda$108/1782133702@7d2c1401
java>
java> boolean contextIsError () { return new Random().nextBoolean(); }
Created method boolean contextIsError()
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
java> IntStream.of(1, 2, 3).forEach( writeLineNo );
1
2
3
java>
```

```
java>
java> IntConsumer writeWarningLineNo = n -> System.out.println(n);
java.util.function.IntConsumer writeWarningLineNo =
Evaluation$cb1ez9x8w0oh3urymia5$$Lambda$107/840823054@5db3dd7e
java>
java> IntConsumer writeErrorLineNo   = n -> System.err.println(n);
java.util.function.IntConsumer writeErrorLineNo =
Evaluation$ukw31ztfbqldmcxa50pg$$Lambda$108/1782133702@7d2c1401
java>
java> boolean contextIsError () { return new Random().nextBoolean(); }
Created method boolean contextIsError()
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
java> IntStream.of(1, 2, 3).forEach( writeLineNo );
1
2
3
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
java> IntStream.of(1, 2, 3).forEach( writeLineNo );
1
2
3
```

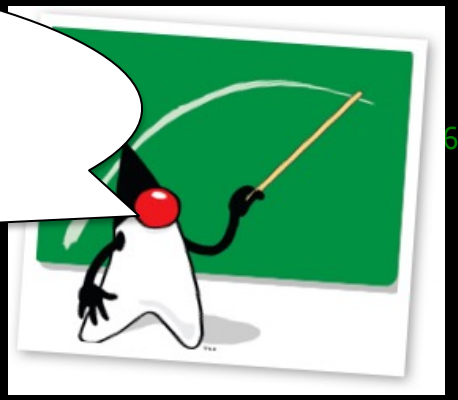
```
java>
java> IntConsumer writeWarningLineNo = n -> System.out.println(n);
java.util.function.IntConsumer writeWarningLineNo =
Evaluation$cb1ez9x8w0oh3urymia5$$Lambda$107/840823054@5db3dd7e
java>
java> IntConsumer writeErrorLineNo   = n -> System.err.println(n);
java.util.function.IntConsumer writeErrorLineNo =
Evaluation$ukw31ztfbqldmcxa50pg$$Lambda$108/1782133702@7d2c1401
java>
java> boolean contextIsError () { return new Random().nextBoolean(); }
Created method boolean contextIsError()
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
java> IntStream.of(1, 2, 3).forEach( writeLineNo );
1
2
3
java>
java>
java>
java>
java> IntStr
1
2
3
```

In addition to the appearance of print methods being passed around in variables, and of method **forEach** being **higher order**, we have the appearance of **higher order** code deciding which print method to pass to **foreach**



```
java>
java> IntConsumer writeWarningLineNo = n -> System.out.println(n);
java.util.function.IntConsumer writeWarningLineNo =
Evaluation$cb1ez9x8w0oh3urymia5$$Lambda$107/840823054@5db3dd7e
java>
java> IntConsumer writeErrorLineNo   = n -> System.err.println(n);
java.util.function.IntConsumer writeErrorLineNo =
Evaluation$ukw31ztfbqldmcxa50pg$$Lambda$108/1782133702@7d2c1401
java>
java> boolean contextIsError () { return new Random().nextBoolean(); }
Created method boolean contextIsError()
java>
java> IntConsumer writeLineNo = contextIsError() ? writeErrorLineNo : writeWarningLineNo;
java.util.function.IntConsumer writeLineNo = Evaluation$smxh3tqkj942za8b75v1$$Lambda$110/10513196
java>
java> IntStream.of(1, 2, 3).forEach( writeLineNo );
1
2
3
ja
ja
java> IntSt
1
2
3
```

But is it really **higher order code**? Are methods really **first-class citizens**? To answer that, let's look at what **first-class functions** means, and what it looks like in FP languages like **Haskell** and **Scala**



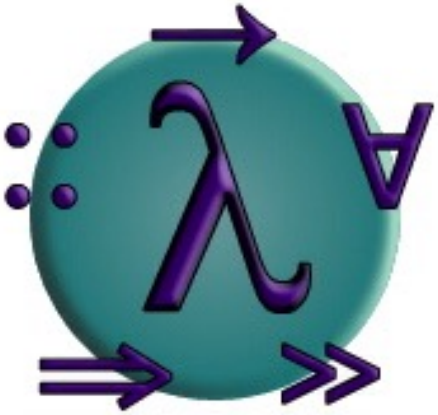
A programming language has **first-class functions** if it treats **functions as first-class citizens**

Functions as first-class citizens:

1.1 **Higher-order functions: passing functions as arguments**
1.2 Anonymous and nested functions
1.3 Non-local variables and closures
1.4 **Higher-order functions: returning functions as results**
1.5 Assigning functions to variables (or storing them in data structures)
1.6 Equality of functions

In languages with first-class functions, **the names of functions** do not have any special status; they are treated like **ordinary variables with a function type**

Let's look at what **function types** and **higher order functions** look like in **Haskell** and **Scala**



Haskell

A Purely Functional Language

featuring static typing, higher-order functions, polymorphism, type classes and monadic effects

Functional Programmers
do it at a higher order!

Haskell

```
lessThan::Int->Int->Bool
lessThan a b = a < b
```

Function Type

Functions

```
greaterThan::Int->Int->Bool
greaterThan a b = a > b
```

Haskell functions can **take functions as parameters** and **return functions as return values**.

A function that does either of these things is called a **higher-order function**.

Function arguments

Function return value

```
choose::Bool->(Int->Int->Bool)->(Int->Int->Bool)->(Int->Int->Bool)
choose condition op1 op2 = if condition then op1 else op2
```

Higher Order Function

```
operator::Int->Int->Bool
operator = choose True lessThan greaterThan
```

Function Variables

```
main = print(operator 3 4)
```

Function application → Applies whatever function is referenced by function variable **operator**


```
// (Int,Int)=>Boolean
val lessThan = (a:Int, b:Int) => a < b
```

Function Type

Functions

```
// (Int,Int)=>Boolean
val greaterThan = (a:Int, b:Int) => a > b
```

A function in Scala is a "**first-class value**". Like any other value, it may be passed as a parameter or returned as a result.

Functions which take other functions as parameters or return them as results are called **higher-order functions**.

Function arguments

Function return value

```
// (Boolean, (Int,Int)=>Boolean, (Int,Int)=>Boolean) => (Int,Int)=>Boolean
val choose = (condition:Boolean, op1:(Int,Int)=>Boolean, op2:(Int,Int)=>Boolean)
        => if (condition) op1 else op2
```

Higher Order Function

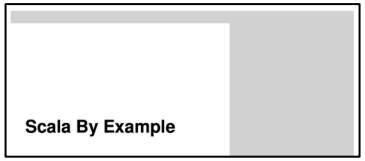
```
// (Int,Int)=>Boolean
val operator = choose(true, lessThan, greaterThan)
```

Function Variables

```
def main = println( operator(3,4) )
```

Function application → Applies whatever function is referenced by the function variable **operator**

Scala



Martin Odersky
in Scala by Example

Scala is a functional language in that functions are first-class values.
Scala is also an object-oriented language in that every value is an object.
It follows that functions are objects in Scala.

For instance, a function from type **String** to type **Int** is represented as an instance of the **trait Function1[String, Int]**

Similar to an **interface** in Java

```
package scala

trait Function1[-A, +B] {
    def apply(x: A): B
}
```

There are also definitions for functions of **all other arities**, i.e. for **each possible number of function parameters** (up to a reasonable limit): **Function1**, **Function2**, … **Function22**.

Scala's function type syntax

(T1, ..., Tn) => S

is simply an abbreviation for the parameterized type

Functionn[T1, ..., Tn,S]

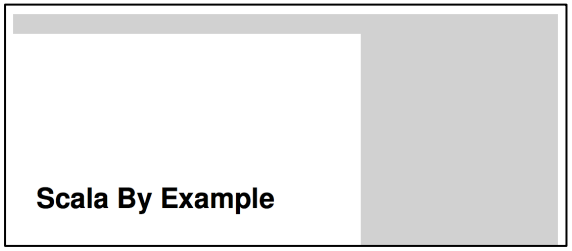
e.g. function type

(String) => Int

is an abbreviation for

Function1[String, Int]

Scala



Martin Odersky
in Scala by Example

Scala uses the **same syntax** `f(x)` for function application, no matter whether `f` is a **method** or a **function object**.

This is made possible by the following convention:

> A **function** application `f(x)` where `f` is an **object** (as opposed to a **method**) is taken to be a shorthand for `f.apply(x)`.

Hence, **the apply method of a function type is inserted automatically** where this is necessary.

e.g. if `f` is a function of type **(String) => Int**
i.e. an object of type **Function1[String, Int]**, then

`f(x)`  sugar for `f.apply(x)`

```
package scala

trait Function1[-A, +B] {
    def apply(x: A): B
}
```

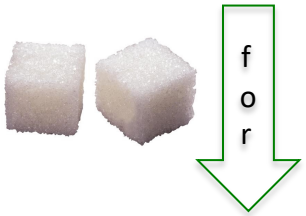
```
// (Int,Int)=>Boolean
val lessThan = (a:Int, b:Int) => a < b
```

Function Type

Scala

`(Int,Int)=>Boolean`  sugar for `Function2[Int,Int,Boolean]`

```
val lessThan = (a:Int, b:Int) => a < b
```



The desugared code looks just like an anonymous inner class in Java

```
val lessThan = new Function2[Int, Int, Boolean] {
    def apply(a: Int, b: Int) = a < b
}
```

The compiler compiles functions down to anonymous classes inside the containing class.

`lessThan(3,4)`  sugar for `lessThan.apply(3,4)`

```
BiFunction<Integer,Integer,Boolean>
lessThan = (a, b) -> a < b;

BiFunction<Integer,Integer,Boolean>
greaterThan = (a, b) -> a > b;

BiFunction<Integer,Integer,Boolean>
choose(Boolean cond,
       BiFunction<Integer,Integer,Boolean> op1,
       BiFunction<Integer,Integer,Boolean> op2) {
    if (cond) return op1; else return op2;
}

BiFunction<Integer,Integer,Boolean>
operator = choose(true, lessThan, greaterThan);

Boolean result = operator.apply(3,4);
```

Functional Interface

Functions

No function types!

Function return value

Function arguments

Higher Order Function

Function Variables

apply!

Function application

Applies whatever function is referenced by the function variable **operator**

Who: **Brian Goetz**, **Architect, Java Language and Libraries**
When: Aug 2011
Mailing List: lambda-dev -- **Technical discussion related to Project Lambda**
Thread: **A peek past lambda**
http://mail.openjdk.java.net/pipermail/lambda-dev/2011-August/003877.html

Simply put, we believe **the best thing we can do for Java developers is to give them a gentle push towards a more functional style of programming.**

We're not going to turn Java into Haskell, nor even into Scala.

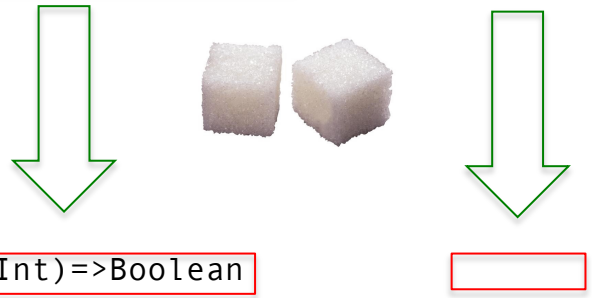
But the direction is clear. Lambda is the down-payment on this evolution, but it is far from the end of the story.
...

I am unwilling to say "Java never will have function types" (though I recognize that **Java may never have function types**.)

Scala

```
val lessThan    = new Function2[Int,Int,Boolean] { def apply(a: Int, b: Int) = a < b }
val greaterThan = new Function2[Int,Int,Boolean] { def apply(a: Int, b: Int) = a > b }
val choose      = (cond:Boolean,
                   op1:Function2[Int,Int,Boolean],
                   op2:Function2[Int,Int,Boolean])
                  => if (cond) op1
                   else op2
val operator    = choose(true, lessThan, greaterThan)
def main        = println( operator.apply(3,4) )
```

Function2[Int,Int,Boolean] .apply



(Int,Int)=>Boolean

```
val lessThan    = (a:Int, b:Int) => a < b
val greaterThan = (a:Int, b:Int) => a > b
val choose      = (cond:Boolean,
                   op1:(Int,Int)=>Boolean,
                   op2:(Int,Int)=>Boolean)
                  => if (cond) op1
                   else op2
val operator    = choose(true, lessThan, greaterThan)
def main        = println( operator(3,4) )
```

Scala

The sugar of function types, plus the 'sugaring away' of the 'apply' method, **hide the OO implementation details of functions**

```
val lessThan    = (a:Int, b:Int) => a < b
val greaterThan = (a:Int, b:Int) => a > b
val choose      = (cond:Boolean,
                   op1:(Int,Int)=>Boolean,
                   op2:(Int,Int)=>Boolean)
                  => if (cond) op1
                     else op2
val operator    = choose(true, lessThan, greaterThan)
def main        = println( operator(3,4) )
```

function types like (Int, Int) => Boolean are obvious and self-explanatory

No need to call 'apply'

No 'function type' sugar and no 'sugaring away' of the 'apply' method: **The OO implementation details of functions are plain to see**

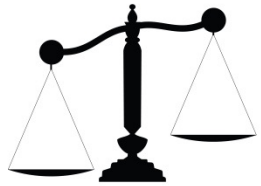
```
BiFunction<Integer,Integer,Boolean> lessThan    = (a, b) -> a < b;
BiFunction<Integer,Integer,Boolean> greaterThan = (a, b) -> a < b;
BiFunction<Integer,Integer,Boolean>
choose(Boolean cond,
       BiFunction<Integer,Integer,Boolean> op1,
       BiFunction<Integer,Integer,Boolean> op2) {
    if (cond) return op1; else return op2;
}
BiFunction<Integer,Integer,Boolean> operator = choose(true, lessThan, greaterThan);
Boolean result = operator.apply(3,4);
```

Some effort required to remember and understand functional interfaces like BiFunction<Integer,Integer,Boolean>

need to call 'apply' or equivalent

Java 8 Predefined Functional Interfaces

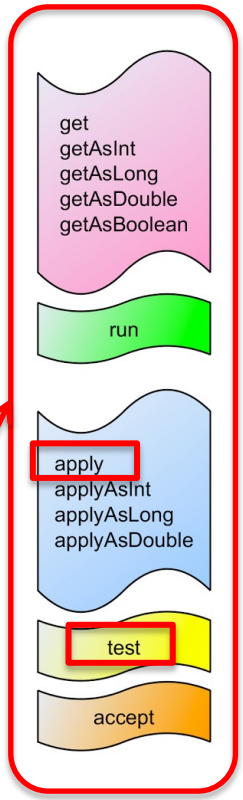
To be fair, **BiFunction<Integer,Integer,Boolean>** , the Java interface we used in the example, can be replaced with the less verbose **BiPredicate<Integer,Integer>**



	→ T	→ int	→ long	→ double	→ boolean	→ void
() →	Supplier<T>	IntSupplier	LongSupplier	DoubleSupplier	BooleanSupplier	Runnable

	→ R	→ int	→ long	→ double	→ boolean	→ void
(T) →	Function<T,R> UnaryOperator<T>	ToIntFunction<T>	ToLongFunction<T>	ToDoubleFunction<T>	Predicate<T>	Consumer<T>
(int) →	IntFunction<R>	IntUnaryOperator	IntToLongFunction	IntToDoubleFunction	IntPredicate	IntConsumer
(long) →	LongFunction<R>	LongToIntFunction	LongUnaryOperator	LongToDoubleFunction	LongPredicate	LongConsumer
(double) →	DoubleFunction<R>	DoubleToIntFunction	DoubleToLongFunction	DoubleUnaryOperator	DoublePredicate	DoubleConsumer

	→ R	→ int	→ long	→ double	→ boolean	→ void
(T, U) →	BiFunction<T,U,R> BinaryOperator<T>	ToIntBiFunction<T,U>	ToLongBiFunction<T,U>	ToDoubleBiFunction<T,U>	BiPredicate<T,U>	BiConsumer<T,U>
(int, int) →		IntBinaryOperator			(T, int) →	ObjIntConsumer<T>
(long, long) →			LongBinaryOperator		(T, long) →	ObjLongConsumer<T>
(double, double) →				DoubleBinaryOperator	(T, double) →	ObjDoubleConsumer<T>



Java 8 Functional Interface Naming Guide
by Esko Luontola, www.orfjackal.net

Note that different types of Functional Interfaces have different invocation methods: **apply** and **accept** (which we have already seen), but also, **test**, **get**, **run**, etc.

Functions in Java 8 are **first class citizens**, **but not in a 'pure' way**,
because **there are no function types**,
which also leads to functions **only being Higher Order in an 'impure' way**.

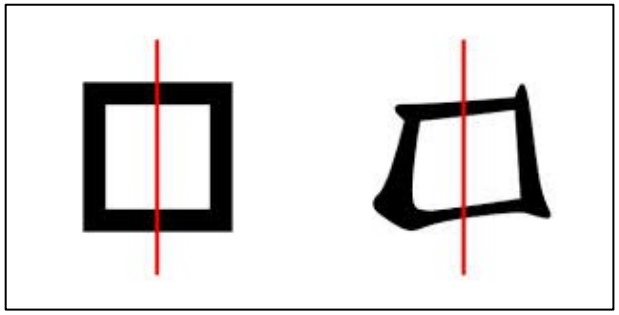
~~pure~~

First Class

~~(T1 → T2) → T3~~

~~pure~~

Higher Order



By that I mean that **there is an asymmetry**

```
twice f n = f(f n)
result = twice (λn.*nn) 3
```

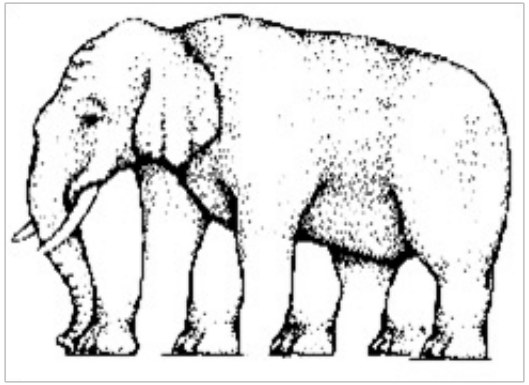
In Java 8, we can pass a **lambda expression** as an **actual parameter**,
but the **formal parameter** has to be a **functional interface**,
so at the call site it looks like we are passing in a function,
but what is passed in is a one-method object,
and to invoke the function, we have to call the object's method

```
Integer result = twice(n -> n * n,3);

Integer twice(UnaryOperator<Integer> f, Integer n)
{
    return f.apply( f.apply(n) );
}
```

There is an asymmetry

the illusion of higher order functions,
i.e. functions being passed around,
is only present at **function/method call sites**

```
Illusion of HoFs
 (T1 → T2) → T3
```



λ

```
n -> n*n
```

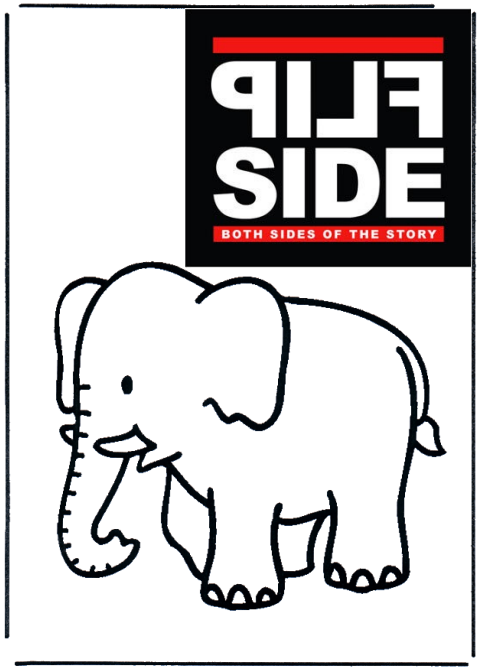
functions

call site

```
UnaryOperator<Integer>
```

objects

flip side



on the flip side, in **function/method bodies**, the illusion is gone:
what are being passed around are objects

In Scala, if a **formal parameter** has a **function type**, e.g. **f** in the following function,

```
def twice(f:Int => Int, n:Int) = …
```

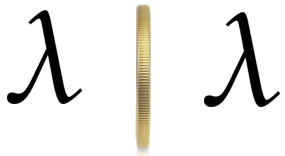
then we can pass a function (a **function literal** or a **function variable**) as an **actual parameter**:

```
val result = twice(n => n * n,3) // passing in a function literal

val square = (n:Int) => n * n

val result = twice(square,3)     // passing in a function variable
```

and on the **flip side**, in the method/function **body**, what is being passed in **is a function** (has a **function type**) and we can **call the function the same way we call a method**:

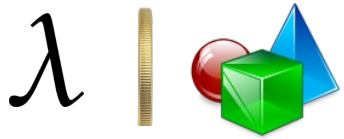
```
def twice(f:Int => Int, n:Int) = f(f(n))
```

So in Scala, the **illusion of higher order code** is supported on **both sides of a Higher Order call**



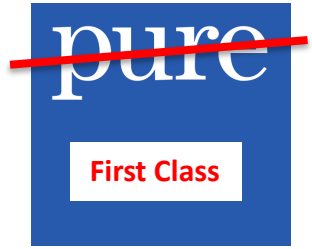
In Java 8, we can use **lambda expressions** to **sweeten** the call sites of higher order methods,

```
Integer result = twice(n -> n * n,3);

Integer twice(UnaryOperator<Integer> f, Integer n)
{
    return f.apply( f.apply(n) );
}
```



but there is **no sugar** to sweeten **method bodies** and **method signatures**

For this reason, Java 8 functions are only **first class citizens** in an **'impure'** way.



We just saw that in Scala, if a **formal parameter** has a **function type**, then we can pass in a **function** as an **actual parameter**

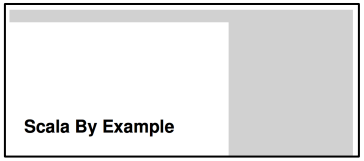
But Scala goes further: **even methods can be passed in where functions are expected!**

```
def twice(f:Int => Int, n:Int) = f(f(n))

def sqr(n:Int) = n * n // define a method, not a function

val result = twice(sqr,3) // the sqr method is promoted to a function!
```

The compiler **promoted our sqr method to a function, so we can pass it to twice**

This is thanks to **Eta-expansion.**

Martin Odersky



*'Eta-expansion converts an expression of **method type** to an **equivalent** expression of **function type**'*

Remember the **α conversion** and **β reduction** of the **λ-calculus**?

η-expansion is one of the two forms of **η-conversion**

η-conversion is adding or dropping of abstraction over a function.

It converts between `λx.fx` and `f` (whenever x does not appear free in f)

η-reduction converts `λx.fx` to `f`

η-expansion converts `f` to `λx.fx`

Scala uses, **η-expansion** to **replace a method reference**
with a function object that wraps the method
and whose apply method takes the parameters of the referenced method
and calls the referenced method with those parameters:

```
twice(sqr,3)
```

for

```
twice(new Function1[Int,Int]{ def apply(x:Int) = sqr(x) }, 3)
```

In Java too we can pass in a method where a function (functional interface) is expected, but in a more verbose way.

We can't just pass in a **method name**

```
Integer twice(UnaryOperator<Integer> f, Integer n)
{
    return f.apply( f.apply(n) );
}

Integer square(Integer n)
{
    return n * n;
}

Integer result = twice(square,3);
```

Cannot resolve symbol 'square'

```
Integer result = twice(square,3);
```

Cannot resolve symbol 'square'

η-expansion

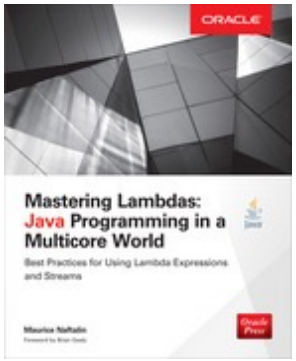
```
Integer result = twice( x -> square(x), 3);
```

But **neither do we have to do the eta expansion ourselves** by wrapping a lambda around the method

Instead, we can use **Method References** (new in Java 8)

Method References

Name	Syntax	Lambda Equivalent
Static	`RefType::staticMethod`	`(args) -> RefType.staticMethod(args)`
Bound Instance	`expr::instMethod`	`(args) -> expr.instMethod(args)`
Unbound Instance	`RefType::instMethod`	`(arg0,rest) -> arg0.instMethod(rest)`
Constructor	`ClsName::new`	`(args) -> new ClsName(args)`




```
Integer result = twice( foo::square, 3);
Integer result = twice( this::square, 3);
```

 for **λ expression**

requests for **η-expansion**

Instance method references
(if **square** is an **instance method**)

```
Integer result = twice( Foo::square, 3);
```

Static method reference
(if **square** is a **static method**)

There is also another reason why the first-class citizenship of Java 8 functions is 'impure'

Functions as first-class citizens:

1.1 Higher-order functions: passing functions as arguments
1.2 Anonymous and nested functions
1.3 Non-local variables and closures
1.4 Higher-order functions: returning functions as results
1.5 Assigning functions to variables (or storing them in data stru
1.6 Equality of functions

We must distinguish between **several types of function equality**

Extensional equality
Two functions are considered equal if they **agree on their outputs** for all inputs

Impractical to implement

Intensional equality
Two functions are considered equal if they have the **same "internal structure"**

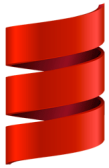
Reference equality

most languages supporting function equality use this

All functions are assigned a unique identifier. Two functions are equal if they have **the same identifier.**
Two separately defined, but otherwise identical function definitions will be considered unequal.

Haskell

Referential equality breaks referential transparency and is therefore not supported in pure languages, such as Haskell



Scala

We saw that functions are objects in Scala, so they have an **equals** method

```
val f = (n:Int) => n * n      f: (Int) => Int =
val g = (n:Int) => n * n      g: (Int) => Int =
f.equals(f)                   res1: Boolean = true
f.equals(g)                   res2: Boolean = false
```

No surprises here

What about Java 8?

If lambda expressions are just **syntactic sugar** for **anonymous inner classes** then they are objects and so they have an **equals method** (inherited from the Object class)

```
java> UnaryOperator<Integer> f = n -> n * n
java> UnaryOperator<Integer> g = n -> n * n
java> f.equals(f)                              java.lang.Boolean res1 = true
java> f.equals(g)                              java.lang.Boolean res2 = false
```

No surprises here

But it turns out that lambda expressions **may or may not** have a **unique identity**, **depending on their implementation**

so `f.equals(f)` may or may not evaluate to true!

There is no Equality of functions in Java 8

That's the other reason why the first-class citizenship of Java 8 functions is '**impure**'

Why is it that ‘lambda expressions **may or may not** have a unique identity’?

Aren’t they objects? Don’t they inherit the **equals** method of the **Object** class?

Who: **Brian Goetz**, **Architect, Java Language and Libraries**
When: Aug 2011
Mailing List: lambda-dev -- **Technical discussion related to Project Lambda**
Thread: **A peek past lambda**
http://mail.openjdk.java.net/pipermail/lambda-dev/2011-August/003877.html

…
lambdas are not objects. I believe **the "lambdas are just objects" position** … **slams the door on** … **potentially useful directions for language evolution**.
…
e.g. I believe that in order to get to **function types** we have to…
…
The lambdas-are-objects view of the world conflicts with this possible future. The lambdas-are-functions view of the world does not, and preserving this flexibility is one of the points in favor of **not burdening lambdas with even the appearance of object-ness**.
…
Lambdas-are-functions opens doors
Lambdas-are-objects closes them
We prefer to see those doors left open
…

Lambdas as Functions

~~Lambdas as Objects~~

Maurice Naftalin

Maurice Naftalin's Lambda FAQ

Your questions answered: all about Lambdas and friends

http://www.lambdafaq.org/

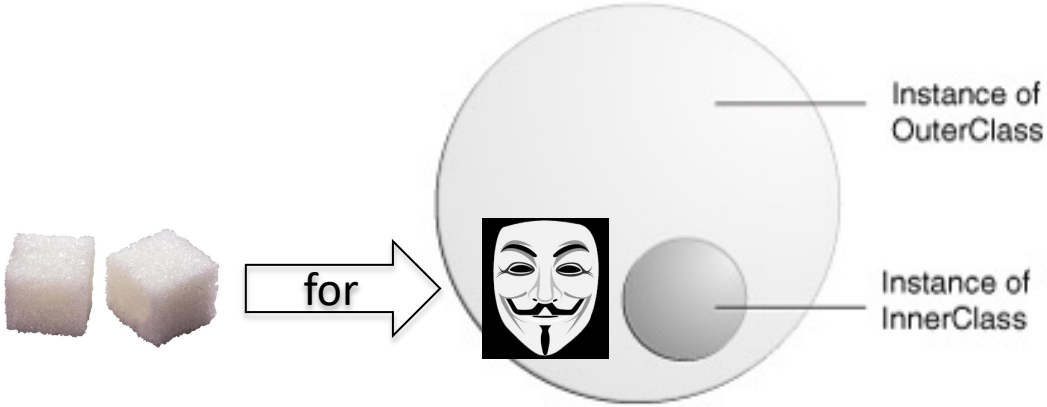
Q: **Are lambda expressions objects?**
A: **Yes**, **with a qualification**: they are instances of object subtypes, but **do not necessarily possess a unique identity.**
…
The question … must be answered on the basis of **how they fit into the Java's type system**, **not** on **how they happen to be implemented at any moment**.

Their status as objects, which stems from the fundamental decision to make them instances of interfaces, **has both positive and negative aspects:**

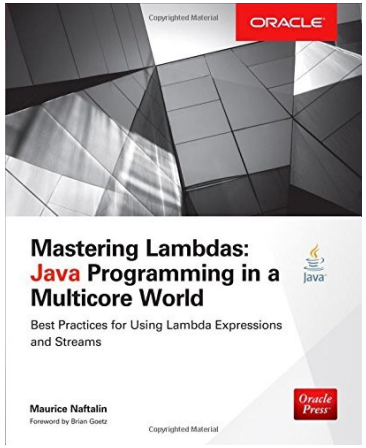
- **lambda expressions inherit the methods of Object.** But note that because lambdas do not necessarily possess a unique identity, **the equals method inherited from Object has no consistent semantics**
- **it enables lambda expressions to fit into the existing type system with relatively little disturbance**;

But if a Java 8 lambda expression is not a full-blown object,

how can it be sugar for an anonymous inner class?

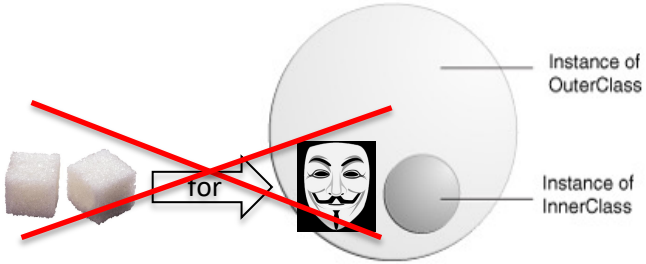


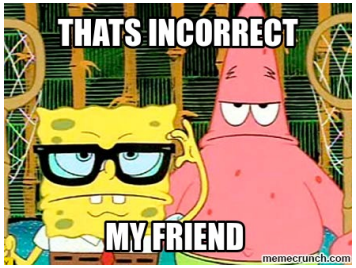
It isn't!!!



Maurice Naftalin

Lambda expressions are sometimes incorrectly called "syntactic sugar" for anonymous inner classes, implying that there is a **simple syntactic transformation between the two**



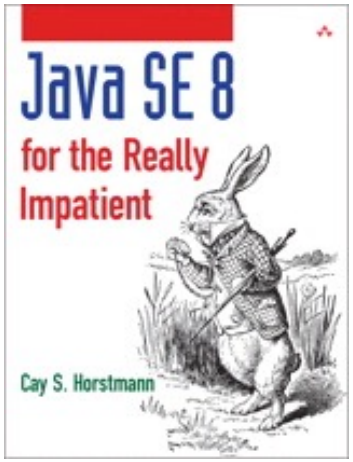


In fact, there are **a number of significant differences**; two in particular are important to the programmer:

1. **An inner class creation expression is guaranteed to create a new object with unique identity**, while **the result of evaluating a lambda expression may or may not have unique identity, depending on the implementation**
2. **An inner class declaration creates a new naming scope**, within which **this** and **super** refer to the current instance of the inner class itself; by contrast, **lambda expressions do not introduce any new naming environment**

```
public class LambdaDemo
{
   public static void main(String[] args)
   {
      new LambdaDemo().doSomething();
   }

   public void doSomething()
   {
      System.out.println(this);
      Runnable r = new Runnable()
         {
            @Override
            public void run()
            {
               System.out.println(this);
            }
         };
      new Thread(r).start();
      new Thread(() -> System.out.println(this)).start();
   }
}
```

any **this** or **super** reference that appears in a lambda body is the **same as in the enclosing scope** because **a lambda doesn't introduce a new scope**, which is not the case with **anonymous classes**



LambdaDemo@15db9742

LambdaDemo$1@293ed848

LambdaDemo@15db9742

anonymous inner class

lambda expression

Same as in the enclosing scope

One question often arises in connection with the rule for interpreting **this**: can a lambda refer to itself?

Remember this definition of factorial?

```
factorial(n): if n=0 then 1 else n * factorial(n – 1)
```

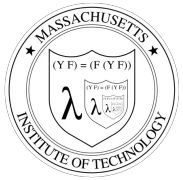
It was recursive, so the λ-calculus version was problematic because it was self-referential

```
fact = λn.if (isZero n) then 1 else * n (fact (pred n))
```

problematic self reference

We eliminated the self-reference by using the Y combinator

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$




```
fact = Y(λf.λn.if (isZero n) then 1 else * n (f(pred n)))
```

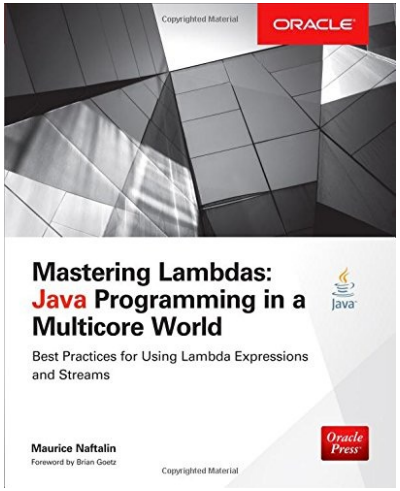
solution: implement recursion using the Y combinator

Is it possible in Java 8 to write a **recursive lambda expression**, e.g. one for **factorial**?

```
public class Factorial
{
    UnaryOperator<Integer> fact = n -> n == 0 ? 1 : n * fact.apply(n-1);
}
```

Illegal self reference

Maurice Naftalin



It is still possible to declare a recursively defined lambda

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

not needed

```
public class Factorial
{
    UnaryOperator<Integer> fact;
    public Factorial()
    {
        fact = n -> n == 0 ? 1 : n * fact.apply(n-1);
    }
}
```

This **idiom** is considered adequate for the relatively unusual occasions on which a **recursive lambda definition** is required.

BONUS

Slides

Slides

#1

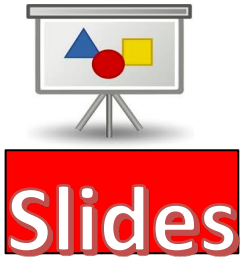
If lambda expressions are not implemented as anonymous inner classes, then how are they implemented?

#2

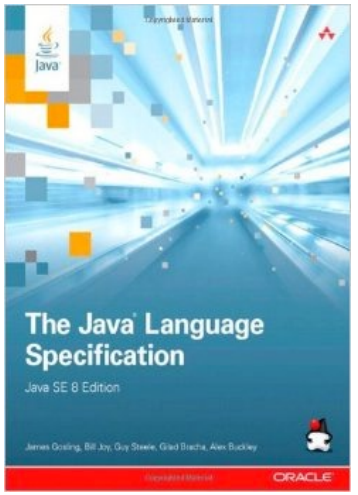
Why are lambda expressions not implemented as anonymous inner classes?

If lambda expressions are not implemented as anonymous inner classes, then how are they implemented?

?



the Java compiler **does not translate** a **lambda expression** into an **anonymous class** during the compilation process



Evaluation of a lambda expression produces an instance of a functional interface

15.27.4 Run-Time Evaluation of Lambda Expressions
At **run time**, evaluation of a lambda expression… **produces a reference to an object**

Let's find out more by comparing the compiler-generated bytecode for the following two classes, which do the same thing, but one using an **anonymous inner class** and the other using a **lambda expression**

```
public class AnonymousInstance
{
    public static void main(String[] args)
    {
        IntStream.of(1,2,3).forEach( new IntConsumer()
        {
            @Override
            public void accept(int n) { System.out.println(n); }
        });
    }
}

public class Lambda
{
    public static void main(String[] args)
    {
        IntStream.of(1, 2, 3).forEach(n -> System.out.println(n));
    }
}
```

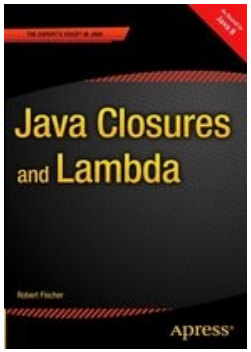
Passes **IntStream.forEach** an anonymous instance of the **IntConsumer functional interface**

Passes **IntStream.forEach** a **lambda expression**

```java
public class AnonymousInstance
{
    public static void main(String[] args)
    {
        IntStream.of(1,2,3).forEach( new IntConsumer()
        {
            @Override
            public void accept(int n) { System.out.println(n); }
        });
    }
}
```

Bytecode of **accept** method of anonymous **IntConsumer** implementation **AnonymousInstance$1**

```
public accept(I)V
L0
  LINENUMBER 13 L0
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  ILOAD 1
  INVOKEVIRTUAL java/io/PrintStream.println (I)V
L1
  LINENUMBER 14 L1
  RETURN
L2
  LOCALVARIABLE this LAnonymousInstance$1; L0 L2 0
  LOCALVARIABLE n I L0 L2 1    MAXSTACK = 2    MAXLOCALS = 2
```



…a **lambda with an inline definition**…What the compiler will do in this case is create a method for you with that implementation. That is called a "**synthetic method**."

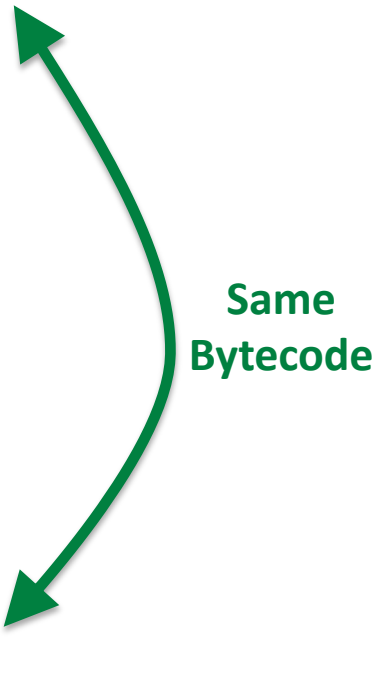
```
public class Lambda
{
    public static void main(String[] args)
    {
        IntStream.of(1, 2, 3).forEach(n -> System.out.println(n));
    }
}
```

Bytecode of the compiler-generated method for the lambda body:
Lambda.lambda$main$0

```
private static synthetic lambda$main$0(I)V
  L0
    LINENUMBER 8 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 0
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
    RETURN
  L1
    LOCALVARIABLE n I L0 L1 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

Unsurprisingly, the bytecode of **AnonymousInstance$1.accept** and **Lambda.lambda$main$0** is essentially the same (prints an integer parameter to **System.out**)

```
public accept(I)V
  L0
    LINENUMBER 13 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 1
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
  L1
    LINENUMBER 14 L1
    RETURN
  L2
    LOCALVARIABLE this LAnonymousInstance$1; L0 L2 0
    LOCALVARIABLE n I L0 L2 1    MAXSTACK = 2    MAXLOCALS = 2


private static synthetic lambda$main$0(I)V
  L0
    LINENUMBER 8 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 0
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
    RETURN
  L1
    LOCALVARIABLE n I L0 L1 0
    MAXSTACK = 2
    MAXLOCALS = 1
```



```java
public class AnonymousInstance
{
    public static void main(String[] args)
    {
        IntStream.of(1,2,3).forEach( new IntConsumer()
        {
            @Override
            public void accept(int n) { System.out.println(n); }
        });
    }
}

public class Lambda
{
    public static void main(String[] args)
    {
        IntStream.of(1, 2, 3).forEach(n -> System.out.println(n));
    }
}
```

We saw the two classes using the same bytecode to print an integer to System.out

The same bytecode is used by the two classes to create the stream and to call the stream's **forEach** method with the **IntConsumer**

Different bytecode is used by each class to create the **IntConsumer** instance

OPERATION	CALLING METHOD	WITH ARGUMENT	AND RETURN TYPE
INVOKESTATIC	IntStream.of	integer array	IntStream
???	???	???	???
??? BYTECODE	??? CREATING AN	??? INSTANCE OF	??? IntConsumer
???	???	???	???
INVOKEINTERFACE	IntStream.forEach	IntConsumer	void

There is a difference in how the IntConsumer instance is created

Bytecode creating **anonymous instance** of **IntConsumer**

```
NEW AnonymousInstance$1
DUP
INVOKESPECIAL AnonymousInstance$1.<init> ()V
```

Instead of generating bytecode to create the object that implements the lambda expression…we describe **a recipe for constructing the lambda, and delegate the actual construction to the language runtime**. That recipe is encoded in the …[arguments]… of an **invokedynamic** instruction.

Bytecode creating an **invokedynamic CallSite** (aka lambda factory), which when invoked, returns an instance of **IntConsumer**

```
INVOKEDYNAMIC accept()Ljava/util/function/IntConsumer; [
  // handle kind 0x6 : INVOKESTATIC

java/lang/invoke/LambdaMetafactory.metafactory(...)Ljava/lang/invoke/CallSite;
  // arguments:
  (I)V,
  // handle kind 0x6 : INVOKESTATIC
  Lambda.lambda$main$0(I)V,
  (I)V
]
```

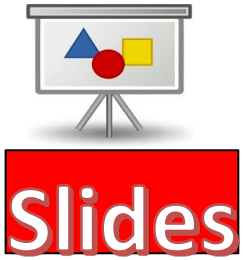
A reference to the **synthetic method** created for the inline lambda expression

Who: **Brian Goetz**, **Architect, Java Language and Libraries**
When: Apr 2012
Page: http://cr.openjdk.java.net/~briangoetz/lambda/**lambda-translation**.html

- There are a number of **strategies** for representing lambda expressions in bytecode
- Each strategy has **pros** and **cons**. E.g. **inner classes** have some **undesirable characteristics** that impact the performance of applications
- **Not committing to a specific strategy** **maximizes flexibility** for future optimization
- Use of **invokedynamic** makes it possible to **defer the selection of a translation strategy until run time**

Why are lambda expressions not implemented as anonymous inner classes?

?

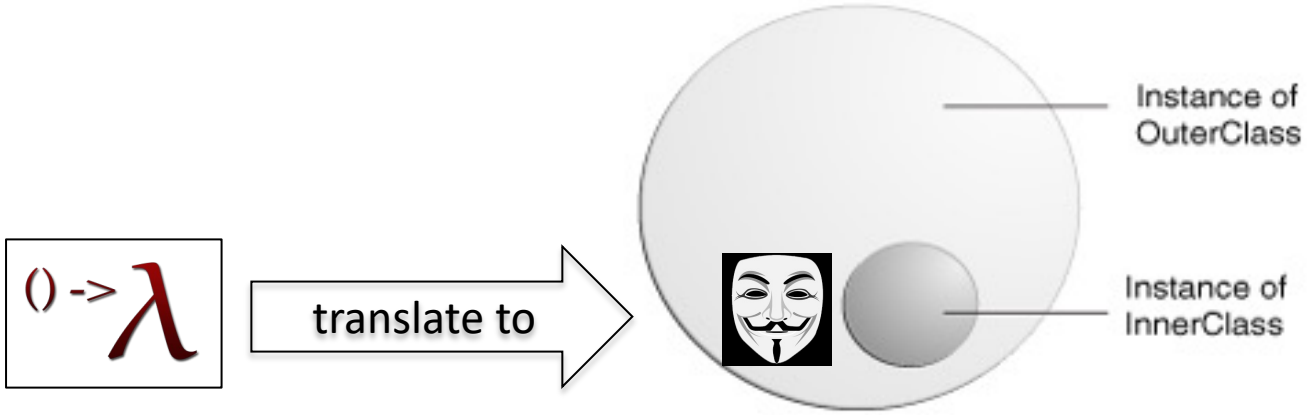


Recap:

- **unlike inner classes, lambda expressions may or may not have unique identity, depending on the implementation**
- **unlike inner class declarations, lambda expressions do not introduce a new naming environment**

Why these differences?

Why not implement **lambda expressions**
as **anonymous inner classes**?



Short Answer: anonymous classes have some **undesirable characteristics** that impact the performance of applications.

A couple of slides from Brian Goetz's 2013 Talk

Lambda expressions for Java

- Big question #1: what is the *type* of a lambda expression?
  - Most languages with lambdas have some notion of a *function type*
    - Java has no concept of function type
    - JVM has no native (unerased) representation of function type in VM type signatures
- Adding function types would create many questions
  - How do we represent functions in VM type signatures?
  - How do we create instances of function-typed variables?
  - How do we deal with variance?
- Want to avoid significant VM changes

Why not “just” use inner classes?

- Translating to inner classes means we inherit most of their problems
  - Performance issues
    - One class per lambda expression
    - Type profile pollution
  - Complicated “comb” lookup
- Whatever we do becomes a *binary representation* for lambdas in Java
  - Would be stuck with it forever
  - Would rather not conflate implementation with binary representation

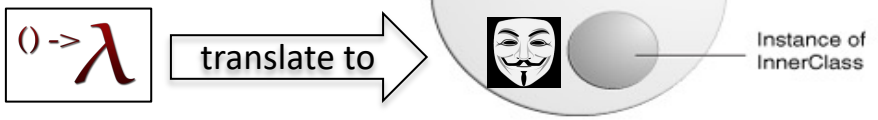
Java 8 Lambdas - A Peek Under the Hood

InfoQ
En | 中文 | 日本 | Fr | Br

http://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood

Why are anonymous inner classes **unsatisfactory**?

If lambdas were translated to anonymous inner classes:



1) you'd have **one new class file for each lambda**, which would be **undesirable** because

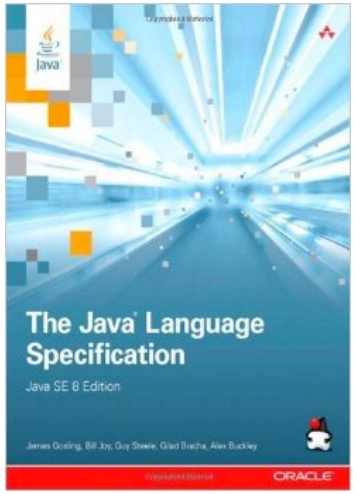
each class file needs to be loaded and verified before being used, which would **impact the startup performance of applications**

as each anonymous inner class would be loaded it would **take up room** in the JVM's meta-space

2) these anonymous inner classes would be **instantiated into separate objects**

as a consequence, anonymous inner classes would **increase the memory consumption of your application**.

Evaluation of a lambda expression produces an instance of a functional interface



15.27.4 Run-Time Evaluation of Lambda Expressions

At run time, evaluation of a lambda expression… **produces a reference to an object**

…

Either a new instance of a class … is allocated and initialized, **or an existing instance** of a class … is referenced.

… offer flexibility to implementations of the Java programming language, in that:

- A **new object need not** be **allocated on every evaluation**.
- Objects produced by different lambda expressions **need not** belong to **different classes** (if the bodies are identical, for example).
- Every object produced by evaluation **need not** belong to the **same class** (captured local variables might be inlined, for example).
- If an "existing instance" is available, it **need not** have been **created at a previous lambda evaluation** (it might have been allocated during the enclosing class's initialization, for example).

these dispensations

offer flexibility

Java 8 Lambdas - A Peek Under the Hood

InfoQ
En | 中文 | 日本 | Fr | Br

So, anonymous inner classes have undesirable characteristics that can impact the performance of your application.

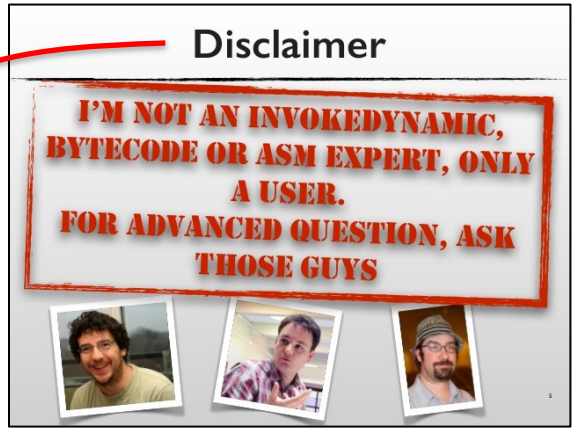
Most importantly, choosing to implement lambdas using anonymous inner class from day one would have **limited:**

- **the scope of future lambda implementation changes**
- as well as the **ability for them to evolve** in line with future JVM improvements

To address the concerns … the Java language and JVM engineers decided to **defer the selection of a translation strategy until run time**.

The **new invokedynamic bytecode instruction** introduced with Java 7 gave them **a mechanism to achieve this in an efficient way**.

I AM NOT EVEN A USER OF INVOKEDYNAMIC, SO DON'T ASK BASIC QUESTIONS EITHER




That's all Folks!