Java Stream API

Streams, Lambdas, and more

Christoffer Fink
christoffer.fink@miun.se

Roadmap

- This is an **interactive** lecture.
- Please interrupt and ask a question if something is confusing.

We will cover:

- Lambdas
- Stream API
- Code examples
- Collector
- More examples

Roadmap

```
public static Map<Character, Integer> charFreq(String s) {
  return IntStream.range(0, s.length())
    .mapToObj(s::charAt)
    .collect(toMap(Function.identity(), c -> 1, Integer::sum));
}

public static void main(String[] args) {
  charFreq(String.join(" ", args))
    .entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEachOrdered(System.out::println);
}
```

What are Streams?

- An alternative “view” of collections.
- Abstract sources of data.
- Similar to IO streams.
- Possibly infinite.

Allowing:

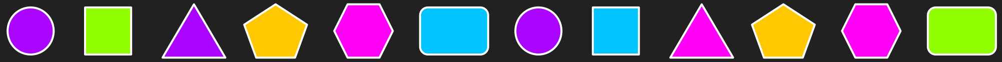
- Lazy sequences.
- A more declarative/functional style.
- Easy parallelism.

What are Streams?

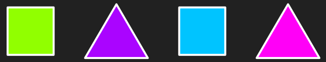
Streams allow you to set up a processing pipeline, similar to text processing on the command line:

```
$ cat book.txt | grep “some phrase” | wc -l
```

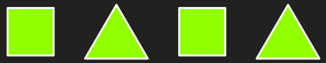
Stream operations visualized, example 1



.filter(isPointy)

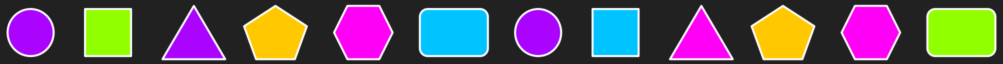


.map(toGreen)

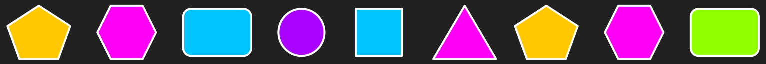


.forEach(Shape::draw)

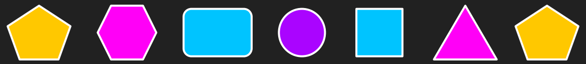
Stream operations visualized, example 2



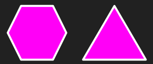
.skip(3)



.limit(7)



.filter(isMagenta)



.count()

2

Streams and Lambdas

- Both new in Java 8.
- Lambdas really shine when combined with streams!
- Streams > Lambdas

Lambdas

- Basically syntactic sugar for creating anonymous inner classes.
- Implement “functional interfaces” on the fly/in-line.

Before Lambdas

Create a new class, either named or anonymous.

```
public class MyRunnable implements Runnable {
        @Override
        public void run() {
                launchMissiles();
        }
}

new Thread(new MyRunnable()).start();
```

```
Runnable r = new Runnable() {
        @Override
        public void run() {
                launchMissiles();
        }
}

new Thread(r).start();
```

Lambdas

```
Runnable r = new Runnable() {
	@Override
	public void run() {
		launchMissiles();
	}
}

new Thread(r).start();
```

```
new Thread(() -> launchMissiles()).start();
```

Lambdas

```
Runnable r = new Runnable() {
        @Override
        public void run() {
                launchMissiles();
        }
}

new Thread(r).start();
```

```
new Thread(() -> launchMissiles()).start();
```

Lambdas

- When/Where can you use lambdas?
- How does the Java compiler know which method you are implementing?

- An interface with exactly one unimplemented method.
- Many generally useful ones in *java.util.function.**
- But also many existing (pre-Java 8).

Lambdas - Syntax

```
(Integer a, String b, Double c) -> {
        double x = Double.parseDouble(b);
        return a + x + c;
};
```

- Types can usually be skipped.
- Parentheses can be skipped for single parameters.
- *return* can be skipped for simple expressions.
- Curly braces can be skipped for simple expressions.

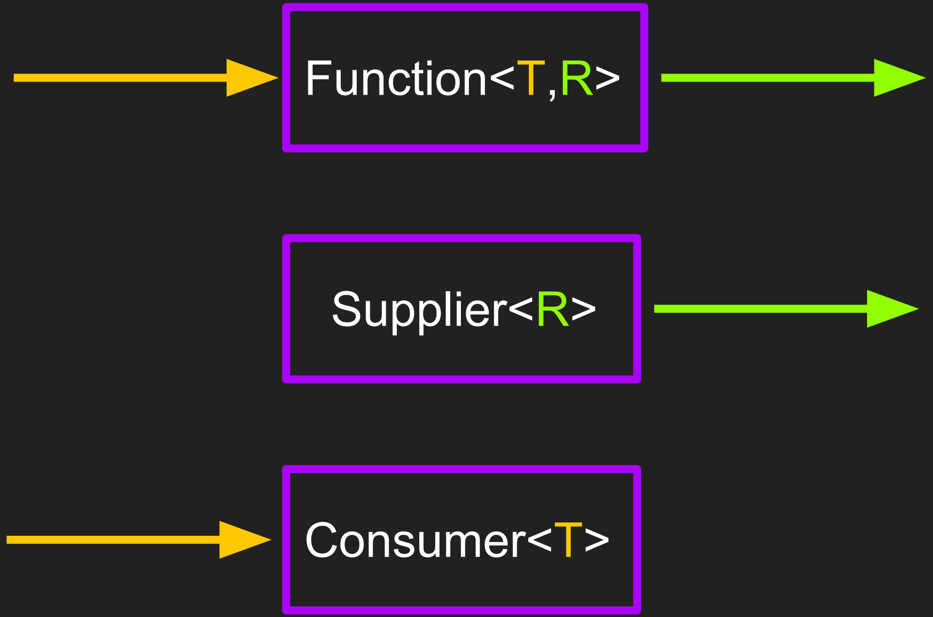
Lambdas - Syntax, quiz

1. (a,b,c) -> a + b + c;
2. (Integer a, String b, Double c) -> a + Double.parseDouble(b) + c;
3. (a,b,c) -> return a + b + c;
4. (a) -> { return a + 13; };
5. a -> { return a + 13; };
6. -> { return 13; };
7. () -> 13;
8. a,b,c -> a + b + c;
9. (Integer a, String b, Double c) ->
   ```
       double x = Double.parseDouble(b);
       return a + x + c;
   ```

Lambdas - Syntax, quiz

```
1.  (a,b,c) -> a + b + c;
2.  (Integer a, String b, Double c) -> a + Double.parseDouble(b) + c;
3.  (a,b,c) -> return a + b + c;
4.  (a) -> { return a + 13; };
5.  a -> { return a + 13; };
6.  -> { return 13; };
7.  () -> 13;
8.  a,b,c -> a + b + c;
9.  (Integer a, String b, Double c) ->
            double x = Double.parseDouble(b);
            return a + x + c;
```

java.util.function.*

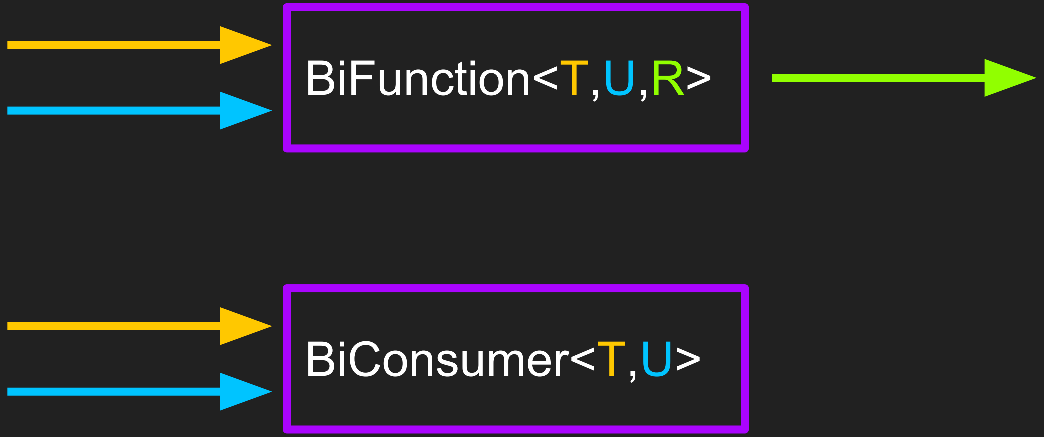


```
y = f.apply(x);

y = s.get();

c.accept(x);
```

java.util.function.*



```
y = f.apply(x,z);
```

```
c.accept(x,z);
```

java.util.function.* - variations

- Predicate<T> extends Function<T,Boolean>
- UnaryOperator<T> extends Function<T,T>
- BinaryOperator<T> extends BiFunction<T,T,T>
- Primitives:
  - LongPredicate
  - IntBinaryOperator
  - DoubleUnaryOperator

Note! These interfaces are **not** actually defined like this. They are only related to each other conceptually, not through inheritance.

java.util.function.* - quiz 1

Which interface in *java.util.function* does a *java.awt.event.ActionListener* correspond to?

java.util.function.* - quiz 1

Which interface in *java.util.function* does a *java.awt.event.ActionListener* correspond to?

Hint: *void actionPerformed(ActionEvent e)*

java.util.function.* - quiz 1

Which interface in *java.util.function* does a *java.awt.event.ActionListener* correspond to?

Hint: *void actionPerformed(ActionEvent e)*

```
button.addActionListener(e -> System.out.println("button pressed"));
```

Lambdas - Method References

If you already have a method that does what you want, you can name it directly.

```
(String s) -> Integer.parseInt(s);
```

```
Integer::parseInt
```

```
(String s) -> s.toUpperCase();
```

```
String::toUpperCase
```

```
(Integer m) -> n.compareTo(m);
```

```
n::compareTo
```

Streams - creating

```
- collection.stream();
- Stream.of(T...);
- Stream.generate(Supplier);
- Stream.iterate(T, UnaryOperator);
- IntStream.range(int, int);
- random.ints(int, int);
- Stream.empty();
- Arrays.stream(T[]);
```

```
- Arrays.asList(1,2,3).stream();
- Stream.of(1,2,3);
- Stream.generate(() -> 13);
- Stream.iterate(0, x -> x+2);
- IntStream.range(1, 100);
- random.ints(1, 100);
- Stream.empty();
- Arrays.stream(args);
```

Stream operations

- Intermediate
  - Don’t do work
  - Return “immediately”
  - Give you back a stream

- Terminal
  - Do work
  - Produce a final value
  - Consume the stream

Streams - terminal operations

- count( ): long
- forEach(Consumer): void
- forEachOrdered(Consumer): void
- max(Comparator): Optional
- min(Comparator): Optional
- reduce(BinaryOperator): Optional
- reduce(T, BinaryOperator): T
- collect(Collector): ???
- findAny( ): Optional
- findFirst( ): Optional
- anyMatch(Predicate)*: boolean
- allMatch(Predicate)*: boolean
- noneMatch(Predicate)*: boolean

* short circuits

Streams - terminal operations

- count( ): long
- forEach(Consumer): void
- forEachOrdered(Consumer): void
- max(Comparator): Optional
- min(Comparator): Optional
- reduce(BinaryOperator): Optional
- reduce(T, BinaryOperator): T
- collect(Collector): ???
- findAny( ): Optional
- findFirst( ): Optional
- anyMatch(Predicate)*: boolean
- allMatch(Predicate)*: boolean
- noneMatch(Predicate)*: boolean

Why does *max* need a *Comparator*, and why does it return an *Optional*?

Streams - intermediate operations

- map*(Function)
- filter(Predicate)
- skip(long)
- limit(long)
- peek(Consumer)
- flatMap*(Function)
- sequential( )
- parallel( )
- distinct( )
- concat(Stream, Stream)
- sorted(Comparator)

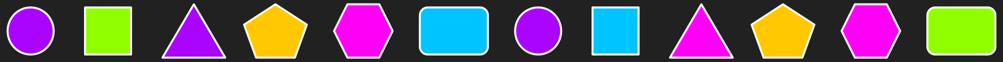
* to*Primitive*

Primitive Streams

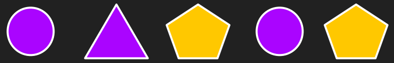
- boxed( ): Stream<Integer>
- asLongStream( ): LongStream
- asDoubleStream( ): DoubleStream
- average( ): OptionalDouble
- map(IntUnaryOperator): IntStream
- mapToLong(IntToLongFunction): LongStream
- mapToObj(IntFunction<R>): Stream<R>
- max( ): OptionalInt
- min( ): OptionalInt
- reduce(IntBinaryOperator): OptionalInt
- reduce(int, IntBinaryOperator): int
- sum( ): int
- summaryStatistics( ): IntSummaryStatistics

- static range(int, int): IntStream
- static rangeClosed(int, int): IntStream

Stream operations visualized, example 3



.filter(isOrange.or(isPurple))



.mapToInt(Shape::numberOfCorners)

0 3 5 0 5

.sum()

13

Stream operations - code example 1, summing

```
int sum = 0;
for(int n : numbers) {
        sum += n;
}
return sum;
```

```
return numbers.stream()
        .mapToInt(Integer::intValue)
        .sum();
```

```
return numbers.stream()
        .reduce(0, Integer::sum);
```

Stream operations - code example 2, find max

```
int max = Integer.MIN_VALUE;
for(int n : numbers) {
        if(n > max) {
                max = n;
        }
}
return max;
```

```
int max = Integer.MIN_VALUE;
for(int n : numbers) {
        max = Math.max(max, n);
}
return max;
```

```
return numbers.stream()
        .max(Integer::compareTo)
        .get();
```

```
return numbers.stream()
        .reduce(Integer.MIN_VALUE, Math::max);
```

```
return numbers.stream()
        .mapToInt(Integer::intValue)
        .max()
        .getAsInt();
```

Stream operations - code example 3, factorial

```
long result = 1;
for(int i = 1; i <= n; i++) {
        result *= i;
}
return result;
```

```
return Stream.iterate(1, x -> x+1)
        .limit(n)
        .reduce(1, (a,b) -> a*b));
```

```
return n <= 1 ? 1 : n*factorial(n-1);
```

```
return IntStream.range(1, n+1)
        .reduce(1, (a,b) -> a*b);
```

Stream operations - quiz

```
List<String> numbers = read();

int sum = 
        .
        .
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.🟪
        .🟨
        .🟩
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .
        .
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .map(l[redacted])
        .[redacted]
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .map(Integer::parseInt)
        .
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .map(Integer::parseInt)
        .reduce(🟪 🟦
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .map(Integer::parseInt)
        .reduce(0, 
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
List<String> numbers = read();

int sum = numbers.stream()
        .map(Integer::parseInt)
        .reduce(0, Integer::sum);
```

Suppose we have a list of strings representing numbers.

What should we put in the blanks to get the sum of all the numbers?

Stream operations - quiz

```
return Stream.iterate(0, n -> n+1)
        .allMatch(n -> n > 5));
```

What does this code do?

java.util.stream.Collectors.*

- toList( ): Collector
- toSet( ): Collector
- joining( ): Collector

- maxBy(Comparator): Collector
- minBy(Comparator): Collector
- reducing(BinaryOperator): Collector
- reducing(T, BinaryOperator): Collector
- toCollection(Supplier): Collector

java.util.stream.Collectors.* - code example 4

```
List<String> strings = //…
List<Integer> integers = new ArrayList<>();

for (String s : strings) {
        integers.add(Integer.parseInt(s));
}
```

```
List<String> strings = //…

List<Integer> integers = strings.stream()
        .map(Integer::parseInt)
        .collect(toList());
```

```
import static java.util.stream.Collectors.toList;
```

java.util.stream.Collectors.*, code example 5

```
List<Integer> integers = //…

String list = integers.stream()
        .map(Integer::toString)
        .collect(joining(“,”));
```

```
import static java.util.stream.Collectors.joining;
```

Character Frequencies

```
public static Map<Character, Integer> charFreq(String s) {
  return IntStream.range(0, s.length())
    .mapToObj(s::charAt)
    .collect(toMap(Function.identity(), c -> 1, Integer::sum));
}

public static void main(String[] args) {
  charFreq(String.join(" ", args))
    .entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEachOrdered(System.out::println);
}
```