

# Java Streams



**Richard Warburton**

Java champion, Author and Programmer

@richardwarburto [www.insightfullogic.com](http://www.insightfullogic.com)

**For loops and iterators are a  
low level, error prone  
construct.**



# Java Streams

A stream is a way of supporting functional-style operations on Collections. In other words aggregate operations that work on sequences of values.



# Overview

Introduce Streams in code

Stream Operations

Collectors

Limitations and General features





# Live Coding Streams



# Intermediate Operations on Streams



# Types of Stream Operations

**Intermediate**

**Everything but the last**

**return Stream<T>**

**filter()**

**Terminal**

**Last in the Pipeline**

**return values**

**toList()**



```
streamOfProducts.filter(product -> product.getWeight() > 20)
```

## Filter

**Remove elements from the Stream that don't match a predicate**



```
streamOfProducts.map(Product::getName)
```

## Map

**Transform elements from one value into another**



## Skip and limit

```
streamOfProducts  
    // Discard next N elements  
    .skip(elementsOnPage * pageNumber)  
  
    // Only keep next N elements  
    .limit(elementsOnPage)
```

# Sorted

```
// Sort Comparable objects with default order  
products.map(Product::getName).sorted()
```

```
// Sort objects with a specified comparator  
Comparator<Product> byName = Comparator.comparing(Product::getName)  
products.sorted(byName)
```



```
streamOfShipments.flatMap(shipment -> shipment.getLightVanProducts().stream())
```

## FlatMap

**Transform elements from one value into zero, one or many values**



# Terminal Operations on Streams



```
productStream.toList()
```

- ◀ Create a List with elements of the Stream
- ◀ List is unmodifiable

```
// Object[]
```

```
productStream.toArray()
```

```
// Product[]
```

```
productStream.toArray(Product[]::new)
```

- ◀ Creates an Object[] Array

- ◀ Pass a function to create the specific array type

- ◀ Normally used with method reference



```
products.anyMatch(
```

```
    prod -> prod.getWeight() > 20);
```

◀ Match family

```
products.noneMatch(
```

```
    prod -> prod.getWeight() > 20);
```

◀ Terminal Operation

◀ Returns a Boolean

```
products.allMatch(
```

```
    prod -> prod.getWeight() > 20);
```

◀ If any / none / all elements match a  
Predicate



```
// max (or min) element given a sort order  
products.max(Comparator.comparingInt(Product::getWeight))  
  
// Side effecting action for each element  
products.forEach(prod -> System.out.println(prod.getName()))  
  
// findFirst (or findAny()) get the element  
products.filter(prod -> prod.getName().contains("Chair")).findFirst()  
  
// Count number of elements in a stream  
products.filter(prod -> prod.getName().contains("Chair")).count()
```



```
streamOfProducts.reduce(0, (acc, product) -> acc + product.getWeight())
```

## Reduce

**Combine elements together using a combining function.**



# Reduce Example

```
// weights of 35, 25  
Stream.of(door, window)  
    .reduce(0, (acc, product) -> acc + product.getWeight());
```

0

(0, door) -> 0 + 35 = 35

(35, window) -> 35 + 25 = 60

60





# Enter the Collector





# Performance and Implementation

# Primitives vs Boxed Numerics



# Primitive Streams

```
strings.mapToInt(String::length)
```

```
IntStream.of(1, 2)
```

```
IntStream.range(start, end)
```



# Primitive Streams

**Performance**  
Improvement over  
boxed numeric  
streams

**Functionality**  
Operations like `sum()`  
for primitives

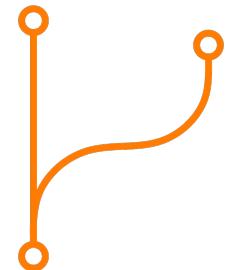
**Specialised**  
`int`, `long` and `double`



# Parallel Streams



**Streams can be run in a parallel mode**



**Run on the common fork-join pool of the JVM**



**Potentially, but not always, a performance improvement**



```
collection.parallelStream()
```

```
stream.parallel()
```

◀ **Explicit but Unobtrusive API**

◀ **Create from a collection**

◀ **Flip an existing stream into parallel mode**



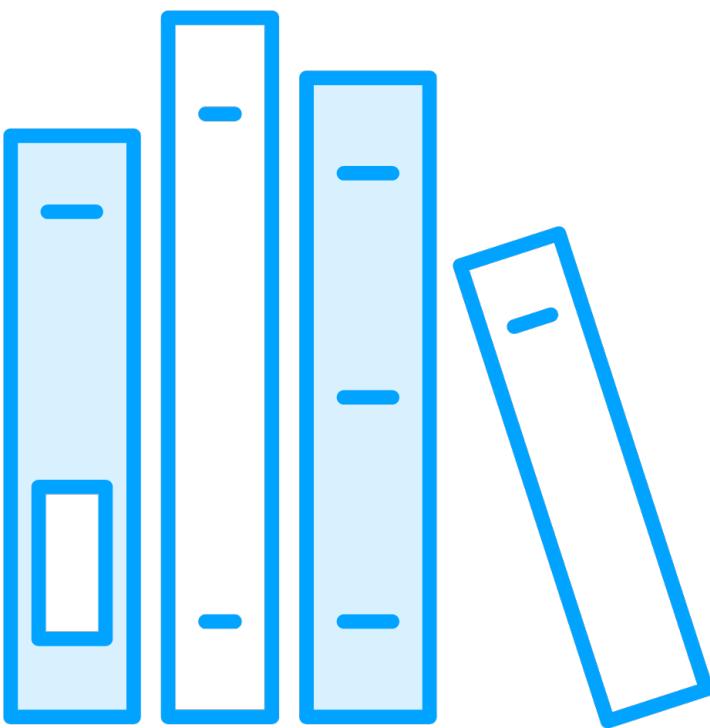
# Conclusion

# Are Streams always better?

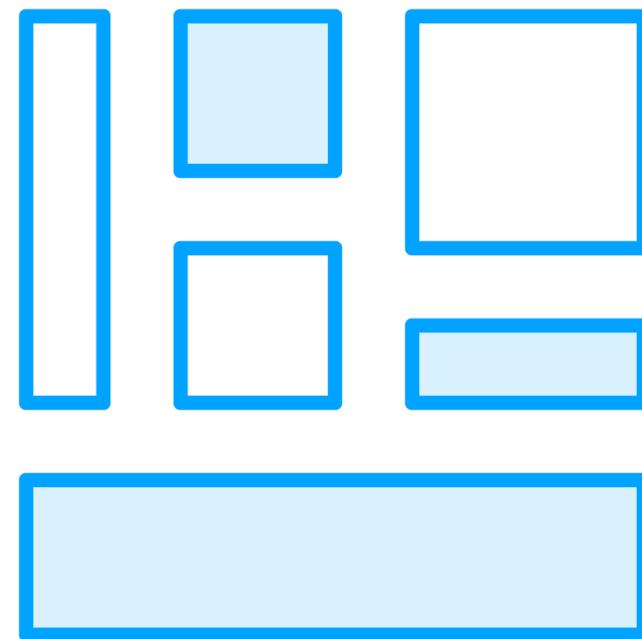
<b>Streams</b>	<b>Loops</b>
<b>High Level construct</b>	<b>Low level construct</b>
<b>Optimized framework</b>	<b>Can be faster</b>
<b>General better readability</b>	<b>Readability is subjective</b>
<b>Some corner cases worse</b>	<b>Nicer with checked Exceptions</b>



# Beyond Streams



**Further Streams  
Learning Material**



**Advanced Collectors**



**Optional API**



# Summary

**Streams are a powerful Abstraction in Java**

**Can replace many for loops and iterators**

**Rely heavily upon Lambda Expressions and Method References.**



**Up Next:**

# **Up Next:**

## **Collection Operations**

