

Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Et cetera

Java lambdas and streams

Charlie Garrod

Michael Hilton

Administrivia

- Homework 5 Best Frameworks available tonight?
- Homework 5c due **Monday** 11:59 p.m.

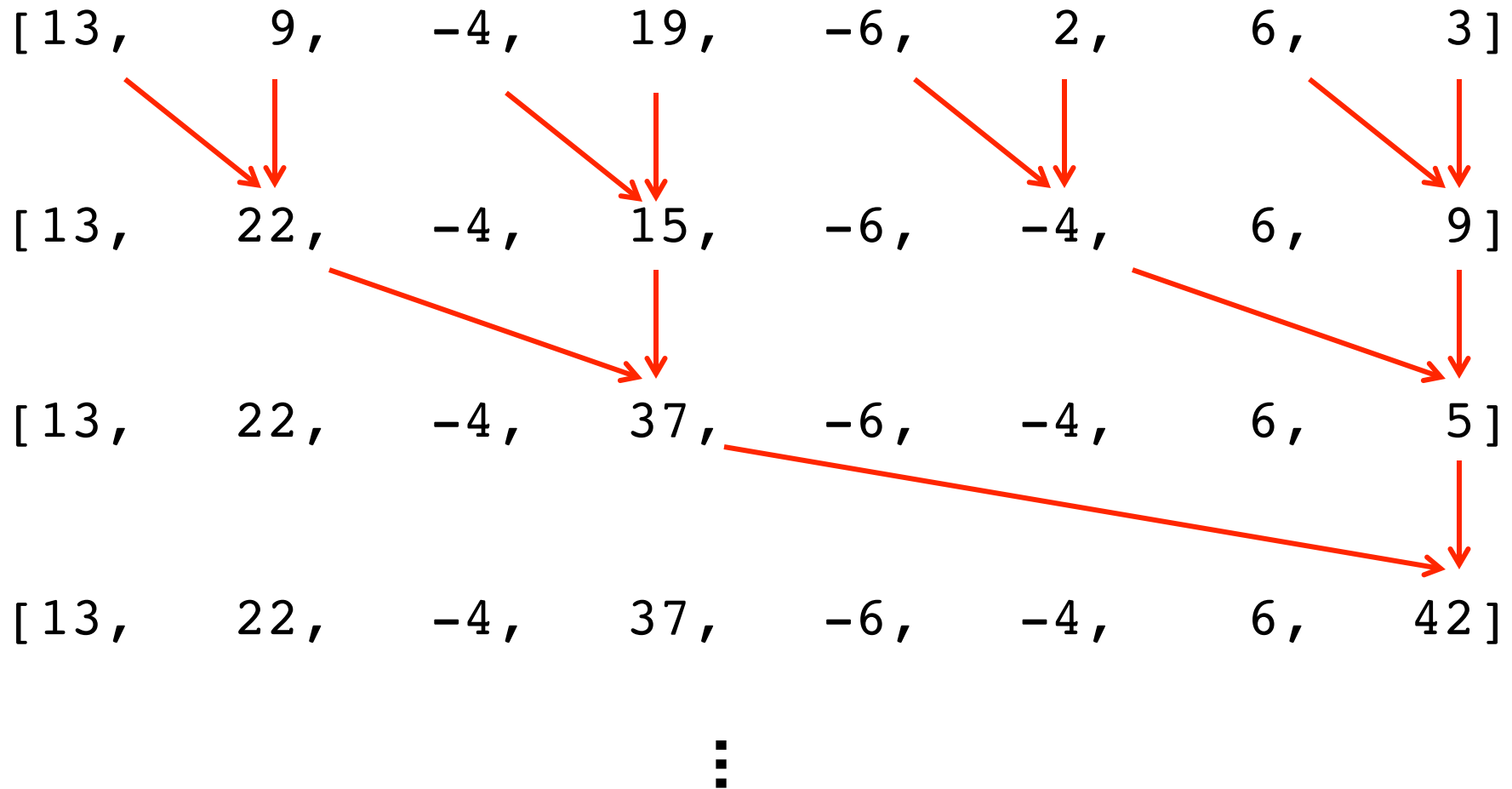
Key concepts from Tuesday

TCP networking in Java: `java.net`

- IP Address: `InetAddress`:
`static InetAddress getByName(String host);`
`static InetAddress getByAddress(byte[] b);`
- Ordinary socket: `Socket`:
`Socket(InetAddress addr, int port);`
`InputStream getInputStream();`
`OutputStream getOutputStream();`
`void close();`
- Server socket: `ServerSocket`:
`ServerSocket(int port);`
`Socket accept();`
`void close();`
...

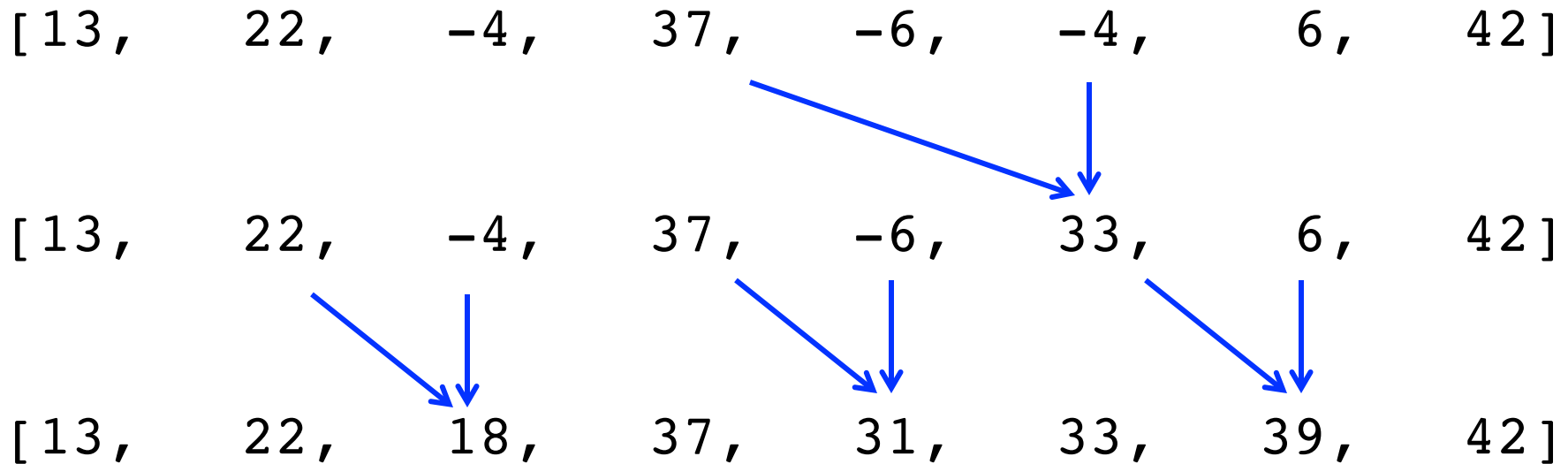
Parallel prefix sums algorithm, upsweep

- Computes the partial sums in a more useful manner



Parallel prefix sums algorithm, downsweep

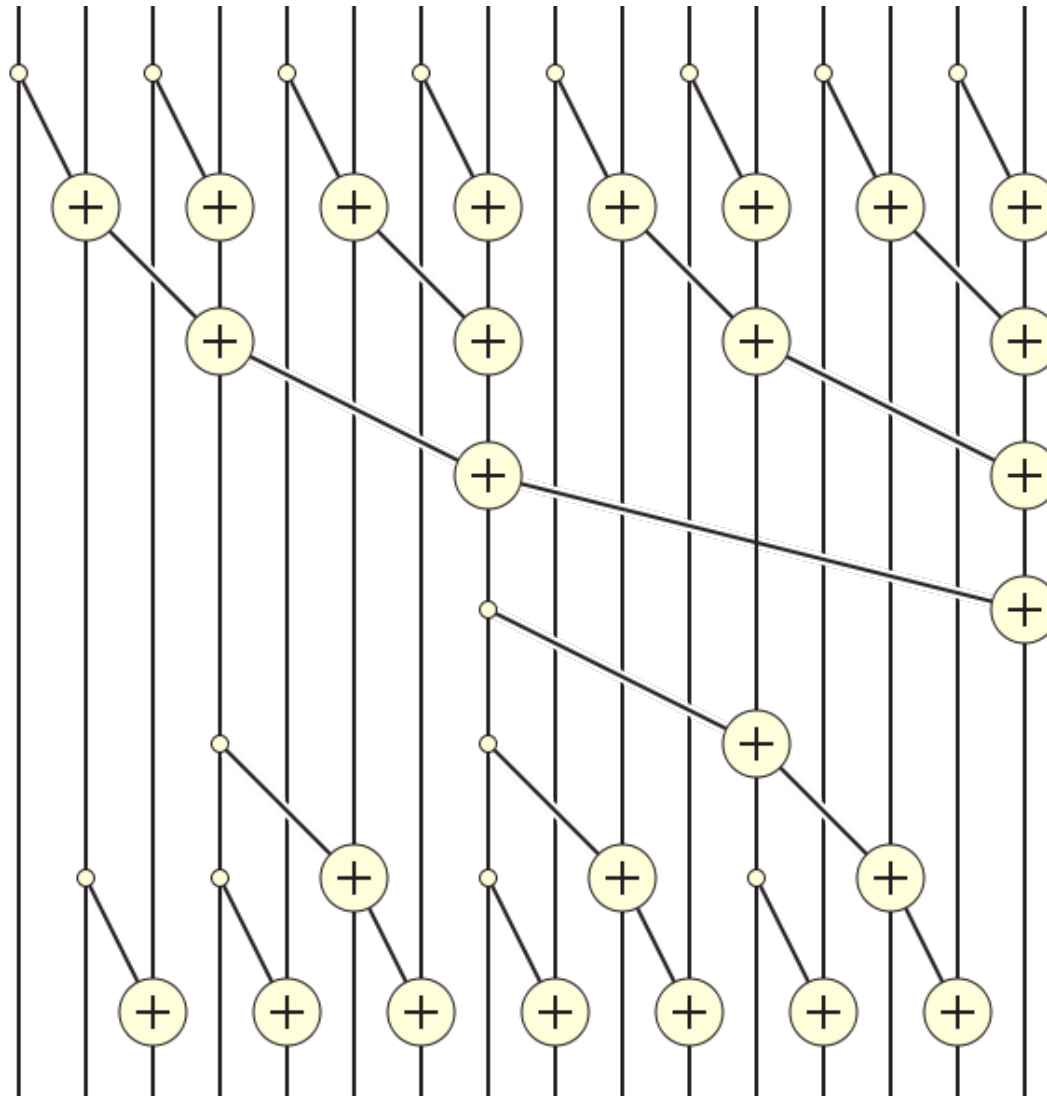
- Now unwinds to calculate the other sums



- Recall, we started with:

[13, 9, -4, 19, -6, 2, 6, 3]

Doubling array size adds two more levels



Upsweep

Downsweep

Fork/Join: computational pattern for parallelism

- **Fork** a task into subtasks
- **Join** the subtasks (i.e., wait for them to complete)
- Subtasks are decomposed recursively
- The `java.util.concurrent.ForkJoinPool` class
 - Implements `ExecutorService`
 - Executes `java.util.concurrent.ForkJoinTask<V>` or `java.util.concurrent.RecursiveTask<V>` or `java.util.concurrent.RecursiveAction`
- The threads in the fork-join pool do *work stealing*

Parallel prefix sums algorithm

- Sequential algorithm: $O(n)$
 - $n-1$ additions
 - Memory access is sequential
 - See `PrefixSumsSequential.java`
- Parallel algorithm: $O(n)$ work, $O(\log n)$ span!
 - About $2n$ useful additions, plus extra additions for the loop indexes
 - Memory access is non-sequential
 - See `PrefixSumsParallel.java`
- The punchline:
 - Don't roll your own
 - Cache and constants matter
 - The *best* parallel implementation was no faster than naïve sequential

Today

- Java lambdas and functional interfaces
- Java streams

Lambdas, briefly

- Term comes from λ -Calculus
 - Everything is a function!
- A lambda (λ) is an *anonymous* function

Does Java have lambdas?

- A. Yes, it's had them since the beginning
- B. Yes, it's had them since anonymous classes (1.1)
- C. Yes, it's had them since Java 8 — the spec says so!
- D. No, never had 'em, never will

Function objects in Java 1.0

```
class StringLengthComparator implements Comparator {  
    private StringLengthComparator() { }  
    public static final StringLengthComparator INSTANCE =  
        new StringLengthComparator();  
  
    public int compare(Object o1, Object o2) {  
        String s1 = (String) o1, s2 = (String) o2;  
        return s1.length() - s2.length();  
    }  
}
```

```
Arrays.sort(words, StringLengthComparator.INSTANCE);
```

Function objects in Java 1.1

```
Arrays.sort(words, new Comparator() {  
    public int compare(Object o1, Object o2) {  
        String s1 = (String) o1, s2 = (String) o2;  
        return s1.length() - s2.length();  
    }  
});
```

"Class Instance Creation Expression" (CICE)

Function objects in Java 5

```
Arrays.sort(words, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

CICE with generics

Function objects in Java 8

```
Arrays.sort(words,  
    (s1, s2) -> s1.length() - s2.length());
```

- They feel like lambdas, they're called lambdas

Lambda syntax

Syntax	Example
parameter -> expression	<code>x -> x * x</code>
parameter -> block	<code>s -> { System.out.println(s); }</code>
(parameters) -> expression	<code>(x, y) -> Math.sqrt(x*x + y*y)</code>
(parameters) -> block	<code>(s1, s2) -> { System.out.println(s1 + "," + s2); }</code>
(parameter decls) -> expression	<code>(double x, double y) -> Math.sqrt(x*x + y*y)</code>
(parameters decls) -> block	<code>(List<?> list) -> { Arrays.shuffle(list); Arrays.sort(list); }</code>

Method references: more succinct than lambdas

- A static method
 - e.g., `Math::cos`
- An unbound instance method (whose receiver is unspecified)
 - e.g., `String::length`
 - The resulting function has an extra argument for the receiver
- A bound instance method of a specific object
 - e.g., `System.out::println`
- A constructor
 - e.g., `Integer::new`, `String[]::new`

No function types in Java, only *functional interfaces*

- Interfaces with only one abstract method
- Optionally annotated with `@FunctionalInterface`
- Some functional interfaces you know
 - `java.lang.Runnable`
 - `java.util.concurrent.Callable`
 - `java.util.Comparator`
 - `java.awt.event.ActionListener`
 - Many, many more in `java.util.function`

Function interfaces in java.util.function

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction

IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>

Some Function<String,Integer>

Description	Code
Lambda	<code>s -> Integer.parseInt(s)</code>
Lambda w/ explicit param type	<code>(String s) -> Integer.parseInt(s)</code>
Static method reference	<code>Integer::parseInt</code>
Constructor reference	<code>Integer::new</code>
Instance method reference	<code>String::length</code>
Anonymous class ICE	<pre>new Function<String, Integer>(){ public Integer apply(String s) { return s.length(); } }</pre>

Java streams

- A stream is a bunch of data objects, typically from a collection, array, or input device, for processing
- Processed by a *pipeline*
 - A single *stream generator* (data source)
 - Zero or more *intermediate stream operations*
 - A single *terminal stream operation*

Stream examples: Iteration

// Iteration over a collection

```
static List<String> stringList = ...;  
stringList.stream()  
    .forEach(System.out::println);
```

// Iteration over a range of integers

```
IntStream.range(0, 10)  
    .forEach(System.out::println);
```

// A mini puzzler: what does this print?

```
"Hello world!".chars()  
    .forEach(System.out::print);
```

Puzzler solution

```
"Hello world!".chars()  
    .forEach(System.out::print);
```

Prints "721011081081113211911111410810033"

The chars method on String returns an IntStream

How do you fix it?

```
"Hello world!".chars()  
    .forEach(x -> System.out.print((char) x));
```

- Now prints "Hello world!"
- Morals:
 - Streams only for object ref types, int, long, and double
 - Type inference can be confusing

Stream examples: mapping, filtering

```
List<String> longStrings = stringList.stream()  
    .filter(s -> s.length() > 42)  
    .collect(Collectors.toList());
```

```
List<String> firstLetters = stringList.stream()  
    .map(s -> s.substring(0,1))  
    .collect(Collectors.toList());
```

```
List<String> firstLetterOfLongStrings =  
    stringList.stream()  
        .filter(s -> s.length() > 42)  
        .map(s -> s.substring(0,1))  
        .collect(Collectors.toList());
```

Stream examples: duplicates, sorting

```
List<String> dupsRemoved = stringList.stream()  
    .map(s -> s.substring(0,1))  
    .distinct()  
    .collect(Collectors.toList());
```

```
List<String> sortedList = stringList.stream()  
    .map(s -> s.substring(0,1))  
    .sorted() // Buffers everything until terminal op  
    .collect(Collectors.toList());
```

Stream examples: bulk predicates

```
boolean allStringHaveLengthThree = stringList.stream()  
    .allMatch(s -> s.length() == 3);
```

```
boolean anyStringHasLengthThree = stringList.stream()  
    .anyMatch(s -> s.length() == 3);
```

Streams are processed lazily

- Data is pulled by terminal operation, not pushed by source
 - Infinite streams are not a problem
- Intermediate operations can be fused
 - Multiple intermediate operations usually don't cause multiple traversals
- Intermediate results usually not stored
 - But there are exceptions (e.g., sorted)

Easy parallelism: `.parallelStream()`

```
List<String> longStrings = stringList.parallelStream()  
    .filter(s -> s.length() > 42)  
    .collect(Collectors.toList());
```

```
List<String> firstLetters = stringList.parallelStream()  
    .map(s -> s.substring(0,1))  
    .collect(Collectors.toList());
```

```
List<String> firstLetterOfLongStrings =  
    stringList.parallelStream()  
        .filter(s -> s.length() > 42)  
        .map(s -> s.substring(0,1))  
        .collect(Collectors.toList());
```

Stream interface is a monster (1/3)

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {  
    // Intermediate Operations  
    Stream<T> filter(Predicate<T>);  
    <R> Stream<R> map(Function<T, R>);  
    IntStream mapToInt(ToIntFunction<T>);  
    LongStream mapToLong(ToLongFunction<T>);  
    DoubleStream mapToDouble(ToDoubleFunction<T>);  
    <R> Stream<R> flatMap(Function<T, Stream<R>>);  
    IntStream flatMapToInt(Function<T, IntStream>);  
    LongStream flatMapToLong(Function<T, LongStream>);  
    DoubleStream flatMapToDouble(Function<T, DoubleStream>);  
    Stream<T> distinct();  
    Stream<T> sorted();  
    Stream<T> sorted(Comparator<T>);  
    Stream<T> peek(Consumer<T>);  
    Stream<T> limit(long);  
    Stream<T> skip(long);
```

Stream interface is a monster (2/3)

// Terminal Operations

```
void forEach(Consumer<T>);           // Ordered only for sequential streams
void forEachOrdered(Consumer<T>);    // Ordered if encounter order exists
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> arrayAllocator);
T reduce(T, BinaryOperator<T>);
Optional<T> reduce(BinaryOperator<T>);
<U> U reduce(U, BiFunction<U, T, U>, BinaryOperator<U>);
<R, A> R collect(Collector<T, A, R>); // Mutable Reduction Operation
<R> R collect(Supplier<R>, BiConsumer<R, T>, BiConsumer<R, R>);
Optional<T> min(Comparator<T>);
Optional<T> max(Comparator<T>);
long count();
boolean anyMatch(Predicate<T>);
boolean allMatch(Predicate<T>);
boolean noneMatch(Predicate<T>);
Optional<T> findFirst();
Optional<T> findAny();
```

Stream interface is a monster (3/3)

// Static methods: stream sources

```
public static <T> Stream.Builder<T> builder();
public static <T> Stream<T> empty();
public static <T> Stream<T> of(T);
public static <T> Stream<T> of(T...);
public static <T> Stream<T> iterate(T, UnaryOperator<T>);
public static <T> Stream<T> generate(Supplier<T>);
public static <T> Stream<T> concat(Stream<T>, Stream<T>);
}
```

In case your eyes aren't glazed yet

```
public interface BaseStream<T, S extends BaseStream<T, S>>
    extends AutoCloseable {
    Iterator<T> iterator();
    Spliterator<T> spliterator();
    boolean isParallel();
    S sequential(); // May have little or no effect
    S parallel();   // May have little or no effect
    S unordered();  // Note asymmetry wrt sequential/parallel
    S onClose(Runnable);
    void close();
}
```

It keeps going: `java.util.stream.Collectors`

```
... toList()  
... toMap(...)  
... toSet(...)  
... reducingBy(...)  
... groupingBy(...)  
... partitioningBy(...)
```

```
    .  
    .  
    .
```

Optional<T>: another way to indicate the absence of a result

It also acts a bit like a degenerate stream

```
public final class Optional<T> {  
    boolean isPresent();  
    T get();  
  
    void ifPresent(Consumer<T>);  
    Optional<T> filter(Predicate<T>);  
    <U> Optional<U> map(Function<T, U>);  
    <U> Optional<U> flatMap(Function<T, Optional<U>>);  
    T orElse(T);  
    T orElseGet(Supplier<T>);  
    <X extends Throwable> T orElseThrow(Supplier<X>) throws X;  
}
```

Stream practice

- Given a `List<String> words`, use streams to:
 - Generate a `List<String>` of all words containing the substring "heat"
 - Determine if any word contains the substring "aoeu" (a boolean)

Stream parallelism: Your mileage may vary

- Consider this for-loop (.96 s runtime; dual-core laptop)

```
long sum = 0;  
for (long j = 0; j < Integer.MAX_VALUE; j++) sum += j;
```
- Equivalent stream computation (1.5 s)

```
long sum = LongStream.range(0, Integer.MAX_VALUE).sum();
```
- Equivalent parallel computation (.77 s)

```
long sum = LongStream.range(0, Integer.MAX_VALUE)  
    .parallel().sum();
```
- Carefully handcrafted parallel code (.48 s)

When to use a parallel stream, loosely speaking

- When operations are independent, and
- Either or both:
 - Operations are computationally expensive
 - Operations are applied to many elements of efficiently splittable data structures
 - Roughly: Number of elements * Cost/element $\gg$ 10,000
- **Always measure before and after parallelizing!**

When not to...

- Use a parallel stream...
- Use a stream...

Summary

- API design: "Fun and easy to learn and use...?"
- When to use a lambda
 - Always, in preference to CICE
- When to use a method reference
 - Almost always, in preference to a lambda
- When to use a stream
 - When it feels and looks right
- When to use a parallel stream
 - Number of elements * Cost/element >> 10,000
- Keep it classy!
 - Java is not a functional language