Principles of Software Construction: Objects, Design, and Concurrency

Lambdas and streams

Josh Bloch Charlie Garrod

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- HW5C (plugins for others' frameworks) due today

Prefix sums (a.k.a. inclusive scan, a.k.a. scan)

- Goal: given array `x[0…n-1]`, compute array of the sum of each prefix of x

```
[ sum(x[0…0]),
  sum(x[0…1]),
  sum(x[0…2]),
  …
  sum(x[0…n-1]) ]
```

- e.g., `x =          [13,  9, -4, 19, -6,  2,  6,  3]`

  prefix sums: `[13, 22, 18, 37, 31, 33, 39, 42]`

institute for SOFTWARE RESEARCH

Parallel prefix sums algorithm, upsweep

Compute the partial sums in a more useful manner

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

[13,  22,  -4,  15,  -6,  -4,   6,   9]

[13,  22,  -4,  37,  -6,  -4,   6,   5]

[13,  22,  -4,  37,  -6,  -4,   6,  42]
```

Parallel prefix sums algorithm, downsweep

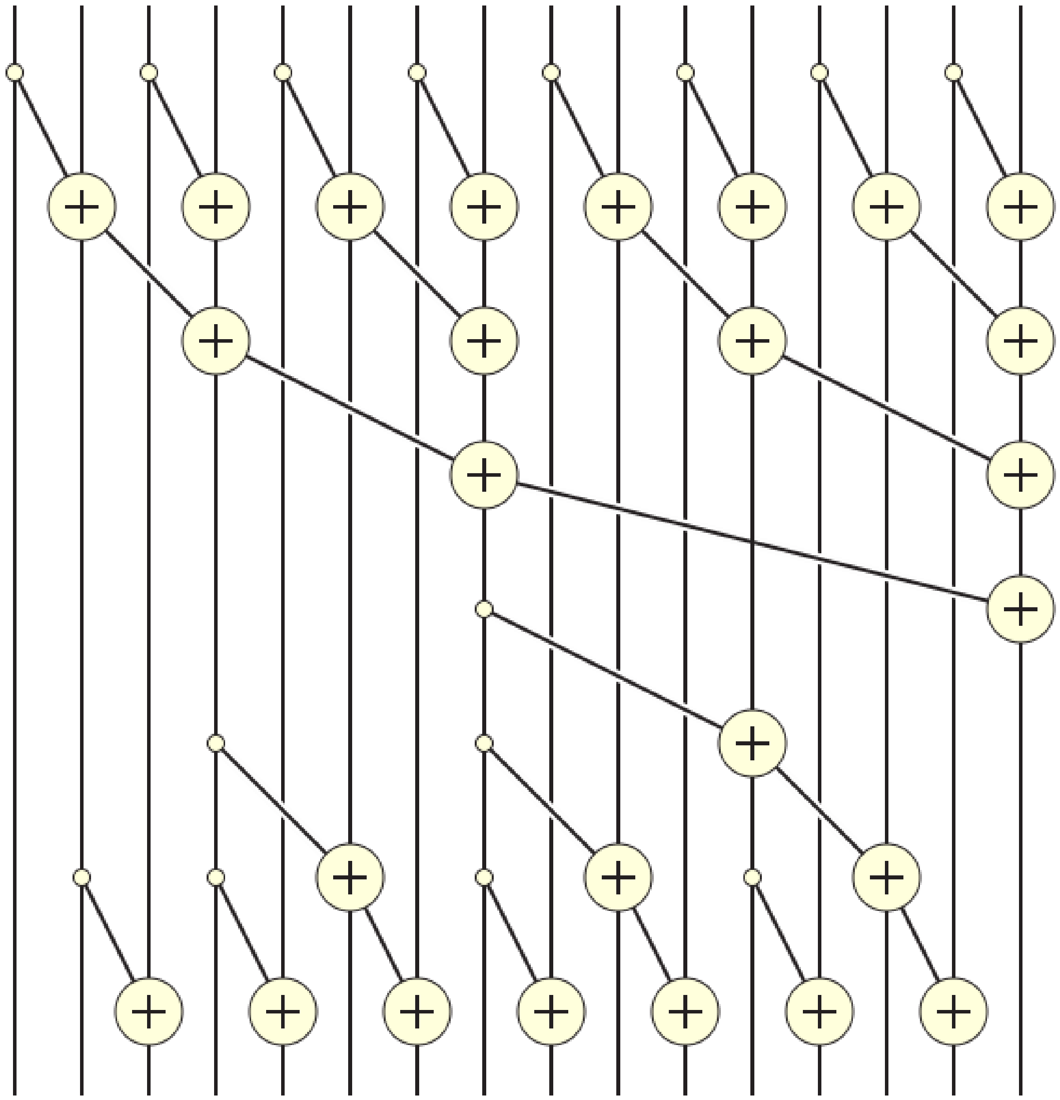
Now unwind to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]

[13,   22,   18,   37,   31,   33,   39,   42]
```

- Recall, we started with:

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

Doubling array size adds two more levels



Recall from Thursday: Fork/join in Java

- 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`
- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See `PrefixSumsParallelArrays.java`
- See `PrefixSumsSequential.java`
  - n-1 additions
  - Memory access is sequential
- For `PrefixSumsSequentialWithParallelWork.java`
  - About 2n useful additions, plus extra additions for the loop indexes
  - Memory access is non-sequential
- The punchline:
  - Don't roll your own. Know the libraries
  - Cache and constants matter

Today's topics

- Two features added in Java 8
  I. **Lambdas**: language feature
  II. **Streams**: library feature
- Designed to work together

I. What is a lambda?

- Term comes from λ-Calculus
  - Formal logic introduced by Alonzo Church in the 1930's
  - Everything is a function!
- **A lambda (λ) is simply an *anonymous* function**
  - A function without a corresponding identifier (name)
- Originally limited to academic languages (e.g., Lisp)
- Popularity exploded in the '90s (JavaScript, Ruby, etc.)
- Now ubiquitous in functional and mainstream languages

When did Java get lambdas?

A. It’s had them since the beginning
B. It’s had them since anonymous classes were added (JDK 1.1, 1997)
C. It’s had them since Java 8 (2014) – the spec says so
D. Never had ’em, never will

Function objects in Java 1.0 (1996)

```
class StringLengthComparator implements Comparator {
    // Singleton
    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 (1997) – anonymous classes

```
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 (2004)

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

Generics made things looks a bit better

Function objects in Java 8 (2014)

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

- They feel like lambdas, and they're called lambdas
  - But they're no more anonymous than 1.1 CICE's!
  - However, method name does not appear in code

Lambda syntax

Syntax	Example
parameter -> *expression*	`x -> x * x`

parameter -> *block*	`i -> { int result = 1;`
	`for (int k = 1; k < i; k++) result *= k;`
	`return result; }`

(*parameters*) -> *expression*	`(x, y) -> Math.sqrt(x*x + y*y)`

(*parameters*) -> *block*	`(i, j) -> { int result = 1;`
	`for (int k = i; k < j; k++) result *= k;`
	`return result; }`

(*parameter decls*) -> *expression*	`(double x, double y) -> Math.sqrt(x*x + y*y)`

(*parameters decls*) -> *block*	`(int i, int j) -> { int result = 1;`
	`for (int k = i; k < j; k++) result *= k;`
	`return result; }`

Java has no function types, only *functional interfaces*

- **Interfaces with only one explicit abstract method**
  - a.k.a *SAM interface* (Single Abstract Method)
- Optionally annotated with `@FunctionalInterface`
  - Do it, for the same reason you use `@Override`
- **A lambda is essentially a functional interface literal**
- Some functional interfaces you already know:
  - `Runnable`, `Callable`, `Comparator`, `ActionListener`
- Many, many more in package `java.util.function`

Java has 43 standard functional interfaces

Luckily, there is a fair amount of structure

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>

The 6 basic standard functional interfaces

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`s -> s.tolowerCase()`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`(i, j) -> i.add(j)`
`Predicate<T>`	`boolean test(T t)`	`c -> c.isEmpty()`
`Function<T,R>`	`R apply(T t)`	`a -> Arrays.asList(a)`
`Supplier<T>`	`T get()`	`Instant.now()`
`Consumer<T>`	`void accept(T t)`	`o -> System.out.println(o)`

Most of the remaining 37 interfaces provide support for primitive types. Use them or pay the price!

A subtle difference between lambdas & anonymous classes

```
class Enclosing {
    Supplier<Object> lambda() {
        return () -> this;
    }

    Supplier<Object> anon() {
        return new Supplier<Object>() {
            public Object get() { return this; }
        };
    }

    public static void main(String[] args) {
        Enclosing enclosing = new Enclosing();
        Object lambdaThis = enclosing.lambda().get();
        Object anonThis = enclosing.anon().get();
        System.out.println(anonThis   == enclosing);  // false
        System.out.println(lambdaThis == enclosing);  // true
    }
}
```

Method references – a more succinct alternative to lambdas

- Lambdas are succinct

```
map.merge(key, 1, (count, incr) -> count + incr);
```

- But *method references* can be more so

```
map.merge(key, 1, Integer::sum);
```

- The more parameters, the bigger the win
  - But parameter names *may* provide documentation
  - If you use a lambda, choose parameter names carefully!

Occasionally, lambdas are more succinct

```
service.execute(() -> action());
```

is preferable to

```
service.execute(GoshThisClassNameIsHumongous::action);
```

Know all five kinds of method references

They all have their uses

Type	Example	Lambda Equivalent*
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now();` `t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>()`
Array Constructor	`int[]::new`	`len -> new int[len]`

The 6 basic functional interfaces redux – method refs

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Lambdas vs. method references – the bottom line

- (Almost) anything you can do with a method reference, you can also do with a lambda
- Method references are *usually* more succinct
- But sometimes lambdas are clearer
- Use your best judgment
  - You can always change your mind
  - Which you use is an implementation detail

II. What is a stream?

- A bunch of data objects (typically from a collection, array, or input device) for bulk data processing
- Processed by a *pipeline*
  - **A single *stream generator* (data source)**
  - **Zero or more *intermediate stream operations***
  - **A single *terminal stream operation***
- Supports mostly-functional data processing
- Enables painless parallelism
  - Simply replace `stream` with `parallelStream`
    - Uses `ForkJoinPool` under the covers
  - You may or may not see a performance improvement

Streams are processed *lazily*

- Data is “pulled” by terminal operation, not pushed by source
  - Infinite streams are not a problem (lazy evaluation)
- Intermediate operations can be fused
  - Multiple intermediate operations usually don’t result in multiple traversals
- Intermediate results typically not stored
  - But there are exceptions (e.g., `sorted`)

Simple stream examples – mapping, filtering, sorting, etc.

```
List<String> longStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());
```

Simple stream examples – mapping, filtering, sorting, etc.

```
List<String> longStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());

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

Simple stream examples – mapping, filtering, sorting, etc.

```
List<String> longStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());

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

List<String> firstLettersOfLongStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .map(s -> s.substring(0,1))
    .collect(Collectors.toList());
```

Simple stream examples – mapping, filtering, sorting, etc.

```
List<String> longStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .collect(Collectors.toList());

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

List<String> firstLettersOfLongStrings = stringList.stream()
    .filter(s -> s.length() > 3)
    .map(s -> s.substring(0,1))
    .collect(Collectors.toList());

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

Simple stream examples – file input

```
// Prints a file, one line at a time
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
    lines.forEach(System.out::println);
}
```

Simple stream examples – file input

```
// Prints a file, one line at a time
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
    lines.forEach(System.out::println);
}

// Prints sorted list of non-empty lines in file (trimmed)
try (Stream<String> lines = Files.lines(Paths.get(fileName))) {
    lines.map(String::trim)
         .filter(s -> !s.isEmpty())
         .sorted()
         .forEach(System.out::println);
}
```

Simple stream examples – bulk predicates

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

Simple stream examples – bulk predicates

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

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

Stream example – the first twenty Mersenne Primes

Mersenne number is a number of the form $2^p - 1$

- If *p* is prime, the corresponding Mersenne number *may be* prime
- If it is, it's a **Mersenne prime**
- Named after Marin Mersenne, a French friar in the early 17th century
- The largest known prime ($2^{82,589,933} - 1$) is a Mersenne prime

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}

public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
            .filter(mersenne -> mersenne.isProbablePrime(50))
            .limit(20)
            .forEach(System.out::println);
}
```

Iterative program to print large anagram groups in a dictionary

Review: you saw this Collections Framework case study

```
public static void main(String[] args) throws IOException {
    File dictionary = new File(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

    Map<String, Set<String>> groups = new HashMap<>();
    try (Scanner s = new Scanner(dictionary)) {
        while (s.hasNext()) {
            String word = s.next();
            groups.computeIfAbsent(alphabetize(word),
                (unused) -> new TreeSet<>()).add(word);
        }
    }

    for (Set<String> group : groups.values())
        if (group.size() >= minGroupSize)
            System.out.println(group.size() + ": " + group);
}
```

Helper function to alphabetize a word

Word nerds call the result an alphagram

```
private static String alphabetize(String s) {
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
```

Streams gone crazy

Just because you can doesn't mean you should!

```
public static void main(String[] args) throws IOException {
    Path dictionary = Paths.get(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

    try (Stream<String> words = Files.lines(dictionary)) {
        words.collect(groupingBy(word -> word.chars().sorted()
                .collect(StringBuilder::new,
                    (sb, c) -> sb.append((char) c),
                    StringBuilder::append).toString()))
                .values().stream()
                    .filter(group -> group.size() >= minGroupSize)
                    .map(group -> group.size() + ": " + group)
                    .forEach(System.out::println);
    }
}
```

A happy medium

Tasteful use of streams enhances clarity and conciseness

```
public static void main(String[] args) throws IOException {
    Path dictionary = Paths.get(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

    try (Stream<String> words = Files.lines(dictionary)) {
        words.collect(groupingBy(word -> alphabetize(word)))
            .values().stream() // Terminal op; create new stream
                .filter(group -> group.size() >= minGroupSize)
                .map(group -> group.size() + ": " + group)
                .forEach(g -> System.out.println(g.size()+": "+g));
    }
}
```

A minipuzzler - what does this print?

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

Puzzler solution

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

Prints `721011081081113211911111410810033`

Why does it do this?

Puzzler solution

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

Prints `721011081081113211911111410810033`

Because `String`'s `chars` method returns an `IntStream`

How do you fix it?

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

Now prints `Hello world!`

Moral

- Streams only for object ref types, `int`, `long`, and `double`
- "Minor primitive types" (`byte`, `short`, `char`, `float`, `boolean`) absent
- Type inference can be confusing
- **Avoid using streams for char processing**

Streams – the bottom line

- Streams are great for many things…
  - But they're not a panacea
- **When you first learn streams, you may want to convert all of your loops. Don't!**
  - It may make your code shorter, but not clearer
- **Exercise judgment**
  - Properly used, streams increase brevity and clarity
  - **Most programs should combine iteration and streams**
- It's not always clear at the outset
  - If you don't know, take a guess and start hacking
  - If it doesn't feel right, try the other approach