Principles of Software Construction: Objects, Design, and Concurrency

Lambdas and streams

Michael Hilton    Bogdan Vasilescu
Administrivia

• Homework 5c due tonight
• Final Exam: Monday, May 6, 2019 05:30 p.m. - 08:30 p.m.
  – Review Session Saturday, May 4th, (Maybe 1-3pm)
Today’s topics

• Two features added in Java 8
  – **Lambdas**: language feature
  – **Streams**: library feature

• Designed to work together
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)
Does Java have lambdas?

A. Yes, it’s had them since the beginning
B. Yes, it’s had them since anonymous classes were added (JDK 1.1, 1997)
C. Yes, it’s had them since Java 8 (2014) – the spec says so
D. No, never had ’em, never will
Function objects in Java 1.0 (1996)

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 (1997)

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();
    }
});

CICE with generics
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
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> | IntUnaryOperator |
| BiFunction<T, U, R> | LongBinaryOperator |
| BinaryOperator<T> | LongConsumer |
| BiPredicate<T, U> | LongFunction<R> |
| BooleanSupplier | LongPredicate |
| Consumer<T> | LongSupplier |
| DoubleBinaryOperator | LongToDoubleFunction |
| DoubleConsumer | LongToIntFunction |
| DoubleFunction<R> | LongUnaryOperator |
| DoublePredicate | ObjDoubleConsumer<T> |
| DoubleSupplier | ObjIntConsumer<T> |
| DoubleToIntFunction | ObjLongConsumer<T> |
| DoubleToLongFunction | Predicate<T> |
| DoubleUnaryOperator | Supplier<T> |
| Function<T, R> | ToDoubleBiFunction<T, U> |
| IntBinaryOperator | ToDoubleFunction<T> |
| IntConsumer | ToIntBiFunction<T, U> |
| IntFunction<R> | ToIntFunction<T> |
| IntPredicate | ToLongBiFunction<T, U> |
| IntSupplier | ToLongFunction<T> |
| IntToDoubleFunction | UnaryOperator<T> |
| IntToLongFunction |
The 6 basic standard functional interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Function Signature</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T apply(T t)</td>
<td>String::toLowerCase</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>T apply(T t1, T t2)</td>
<td>BigInteger::add</td>
</tr>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>boolean test(T t)</td>
<td>Collection::isEmpty</td>
</tr>
<tr>
<td>Function&lt;T,R&gt;</td>
<td>R apply(T t)</td>
<td>Arrays::asList</td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>T get()</td>
<td>Instant::now</td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>void accept(T t)</td>
<td>System.out::println</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter -&gt; expression</td>
<td>x -&gt; x * x</td>
</tr>
<tr>
<td>parameter -&gt; block</td>
<td>s -&gt; { System.out.println(s); }</td>
</tr>
<tr>
<td>(parameters) -&gt; expression</td>
<td>(x, y) -&gt; Math.sqrt(x<em>x + y</em>y)</td>
</tr>
<tr>
<td>(parameters) -&gt; block</td>
<td>(s1, s2) -&gt;</td>
</tr>
<tr>
<td></td>
<td>{ System.out.println(s1 + &quot;,&quot; + s2); }</td>
</tr>
<tr>
<td>(parameter decls) -&gt; expression</td>
<td>(double x, double y) -&gt; Math.sqrt(x<em>x + y</em>y)</td>
</tr>
<tr>
<td>(parameters decls) -&gt; block</td>
<td>(List&lt;??&gt; list) -&gt;</td>
</tr>
<tr>
<td></td>
<td>{ Arrays.shuffle(list); Arrays.sort(list); }</td>
</tr>
</tbody>
</table>
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
  
  \( \text{map.merge(key, 1, (count, incr) \to count + incr)} \);

• But \textit{method references} can be more so
  
  \( \text{map.merge(key, 1, \text{Integer::sum})} \);

• The more parameters, the bigger the win
  
  – But parameter names \textit{may} provide documentation
  
  – If you use a lambda, choose parameter names carefully!
Occasionally, lambdas are more succinct

\[
\text{service.execute}(() \to \text{action}());
\]

is preferable to

\[
\text{service.execute}(\text{GoshThisClassNameIsHumongous}::\text{action});
\]
Know all five kinds of method references

*They all have their uses*

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Lambda Equivalent*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static</strong></td>
<td>Integer::parseInt</td>
<td>str -&gt; Integer.parseInt(str)</td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>Instant.now()::isAfter</td>
<td>Instant then = Instant.now(); t -&gt; then.isAfter(t)</td>
</tr>
<tr>
<td><strong>Unbound</strong></td>
<td>String::toLowerCase</td>
<td>str -&gt; str.toLowerCase()</td>
</tr>
<tr>
<td><strong>Class Constructor</strong></td>
<td>TreeMap&lt;K,V&gt;::new</td>
<td>() -&gt; new TreeMap&lt;K,V&gt;()</td>
</tr>
<tr>
<td><strong>Array Constructor</strong></td>
<td>int[]::new</td>
<td>len -&gt; new int[len]</td>
</tr>
</tbody>
</table>
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
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`
  - 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
- 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());

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);
}

// 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 allStringHaveLengthThree = 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

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

```java
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*

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

```java
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, then create new stream
                .filter(group -> group.size() >= minGroupSize)
                .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 explanation

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

Prints 721011081081113211911111410810033

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 are missing
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
Use caution making streams parallel

*Remember our Mersenne primes program?*

```java
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);
}
```

Runs in 12.5s on my quad-core, 8-thread core i7
How fast do you think this program runs?

```java
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))
        .parallel()
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```
How fast do you think this program runs?

```java
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))
        .parallel()
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

Very, very slowly. I gave up after half an hour.
Why did the program run so slowly?

- **The streams library has no idea how to parallelize it**
  - And the heuristics fail miserably
- In the *best* case, parallel is unlikely to help if:
  - Stream source is `Stream.iterate`, or
  - Intermediate limit operation is used
- This *isn’t* the best case
  - Default strategy for `limit` computes excess elements
  - Each Mersenne prime takes twice as long to compute as last one
- **Moral:** do not parallelize indiscriminately!
What *does* parallelize well?

- Arrays, ArrayList, HashMap, HashSet, ConcurrentHashMap, int and long ranges...

- What do these sources have in common?
  - Predictably splittable
  - Good *locality of reference*

- Terminal operation also matters
  - Must be quick, or easily parallelizable
  - Best are *reductions*, e.g., min, max, count, sum
  - Collectors (AKA *mutable reductions*) not so good

- Intermediate operations matter too
  - Mapping and filtering good, limit bad
Example – number of primes $\leq n$, $\pi(n)$

```java
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

Takes 31s to compute $\pi(10^8)$ on my machine
Example – number of primes ≤ \( n \), \( \pi(n) \)

```java
static long pi(long n) {
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

In parallel, it takes 9.2s, which is 3.7 times as fast!
parallel() is merely an optimization

• Optimize Judiciously [EJ Item 67]
• Don’t parallelize unless you can prove it maintains correctness
• Don’t parallelize unless you have a good reason to believe it will help
• Measure performance before and after
Summary

• 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
  – When you’ve demonstrated that it’s a performance win