Functional Constructs in Java 8: Lambdas and Streams

Josh Bloch Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 6 due Thursday 11:59 pm
- Final exam Tuesday, May 3, 5:30-8:30 pm, PH 100
- Final review session Sun. May, 1, 7-9 pm, DH 1112

Key concepts from Thursday

- Transactions execute concurrently
- And may abort before completion
- But they appear to execute serially & to completion
- Greatly simplify building reliable, persistent, consistent distributed systems

What is a lambda?

- Term comes from λ-Calculus
  - Formal logic introduced by Alonzo Church in the 1930's
  - Everything is a function!
  - Equivalent in power and expressiveness to Turing Machine
  - Church-Turing Thesis, ~1934
- A lambda is 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 (1.1)
C. Yes, it's had them since 8.0 — 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, and they're called lambdas
  - But they're no more anonymous than 1.1 CICE's!
  - Method has name, class does not
  - But method name does not appear in code ☺

No function types in Java, only *Functional Interfaces*

- Interfaces with only one *explicit* abstract method
  - AKA *SAM interface* (Single Abstract Method)
- Optionally annotated with `@FunctionalInterface`
  - Do it, for the same reason you use `@Override`
- Some Functional Interfaces you know
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
  - `java.util.Comparator`
  - `java.awt.event.ActionListener`
  - Many, many more in package `java.util.function`

Lambda Syntax

Syntax	Example
parameter -> expression	`x -> 2 * x`
parameter -> block	`s -> { System.out.println(s); }`
(parameters) -> expression	`(x, y) -> x*x - y*y`
(parameters) -> block	`(s1, s2) ->` `  { System.out.println(s1 + "," + s2); }`
(parameter decls) -> expression	`(double x, double y) -> x*x - y*y`
(parameters decls) -> block	`(List<?> lst) ->` `  { Arrays.shuffle(lst); Arrays.sort(lst); }`

Method references – a more succinct alternative to lambdas

- An instance method of a particular object (*bound*)
  - `objectRef::methodName`
- An instance method, whose receiver is unspecified (*unbound*)
  - `ClassName::instanceMethodName`
  - The resulting function has an extra argument for the receiver
- A static method
  - `ClassName::staticMethodName`
- A constructor
  - `ClassName::new`

What is a stream?

- A bunch of data objects, typically from a collection, array, or input device
- Typically processed by a *pipeline*
  - A single *stream generator* (data source)
  - Zero or more *intermediate stream operations*
  - A *terminal stream operation*
- Supports mostly-functional data processing
- Enables painless parallelism
  - Simply replace `stream` with `parallelStream`

Stream examples (1)

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

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

// Puzzler: what does this print?
"Hello world!".chars()
      .forEach(System.out::print);

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

Stream examples (2)

```
try (Stream<String> linesInFile =
   Files.lines(Paths.get(fileName))) {
     linesInFile.forEach(System.out::println);
}
```

Stream examples (3)

```
boolean allMatch = stringList.stream()
      .allMatch(s -> s.length() == 3);
System.out.println(allMatch);

boolean anyMatch = stringList.stream()
      .anyMatch(s -> s.length() == 3);
System.out.println(anyMatch);
```

Stream examples(4)

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

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

List<String> filteredMappedList =
      stringList.stream()
            .filter(s -> s.length() > 4)
            .map(s -> s.substring(0,1))
            .collect(Collectors.toList());
System.out.println(filteredMappedList);
```

Stream examples (5)

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

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

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 typically don’t result in multiple traversals
- Intermediate results typically not stored
  - But there are exceptions (e.g., sorted)

A simple parallel stream example

- 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();
```

- Fastest handcrafted parallel code I could write (.48 s)
  - You don't want to see the code. It took hours.

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
- Always measure before and after parallelizing!
  - Jackson's third law of optimization

When to use a parallel stream – in detail

- Consider `s.parallelStream().operation(f)` if
  - `f`, the per-element function, is independent
    - i.e., computation for each element doesn't rely on or impact any other
  - `s`, the source collection, is efficiently splittable
    - Most collections, and `java.util.SplittableRandom`
    - NOT most I/O-based sources
  - Total time to execute sequential version roughly > 100µs
    - "Multiply N (number of elements) by Q (cost per element of f), guestimating Q as the number of operations or lines of code, and then checking that N*Q is at least 10,000.
      If you're feeling cowardly, add another zero or two."—DL
    - For details: http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

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>);
void forEachOrdered(Consumer<T>);
java.lang.Object[] toArray();
<A> A[] toArray(IntFunction<A[]>);
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 Op!
<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 (2/3)

```
 // 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();
 S parallel();
 S unordered();
 S onClose(Runnable);
 void close();
}
```

`Optional<T>` – a third (!) 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;
}
```

Summary

- When to use a lambda
  - Always, in preference to CICE
- When to use a method ref
  - 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