Principles of Software Construction: Objects, Design, and Concurrency

Part 4: Et cetera

Design case study: Java lambdas and streams

Charlie Garrod       Chris Timperley
Administrivia

- Homework 5 Best Frameworks available today
- Homework 5c due Tuesday, 11:59 p.m.
Key concepts from Tuesday
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 $x$
  
  $[\text{sum}(x[0…0]),\text{sum}(x[0…1]),\text{sum}(x[0…2]),...\text{sum}(x[0…n-1]) ]$

- e.g., $x = [13, 9, -4, 19, -6, 2, 6, 3]$
  
  prefix sums: $[13, 22, 18, 37, 31, 33, 39, 42]$
Parallel prefix sums algorithm, \textit{upsweep}

Compute the partial sums in a more useful manner

\[
\begin{array}{cccccccc}
13 & 9 & -4 & 19 & -6 & 2 & 6 & 3 \\
13 & 22 & -4 & 15 & -6 & -4 & 6 & 9 \\
13 & 22 & -4 & 37 & -6 & -4 & 6 & 5 \\
13 & 22 & -4 & 37 & -6 & -4 & 6 & 42 \\
\end{array}
\]
Parallel prefix sums algorithm, **downsweep**

Now unwind to calculate the other sums

\[
\begin{bmatrix}
13 & 22 & -4 & 37 & -6 & -4 & 6 & 42 \\
13 & 22 & -4 & 37 & -6 & 33 & 6 & 42 \\
13 & 22 & 18 & 37 & 31 & 33 & 39 & 42 \\
\end{bmatrix}
\]

- Recall, we started with:

\[
\begin{bmatrix}
13 & 9 & -4 & 19 & -6 & 2 & 6 & 3 \\
\end{bmatrix}
\]
Doubling array size adds two more levels

Upsweep

Downsweep
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

- 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

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

- They feel like lambdas, they’re called lambdas
# 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;  {  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;  {  Arrays.shuffle(list);  Arrays.sort(list);  } }</td>
</tr>
</tbody>
</table>
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.langRunnable`
  - `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>

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lambda</td>
<td>s -&gt; Integer.parseInt(s)</td>
</tr>
<tr>
<td>Lambda w/ explicit param type</td>
<td>(String s) -&gt; Integer.parseInt(s)</td>
</tr>
<tr>
<td>Static method reference</td>
<td>Integer::parseInt</td>
</tr>
<tr>
<td>Constructor reference</td>
<td>Integer::new</td>
</tr>
<tr>
<td>Instance method reference</td>
<td>String::length</td>
</tr>
<tr>
<td>Anonymous class ICE</td>
<td>new Function&lt;String, Integer&gt;(){</td>
</tr>
<tr>
<td></td>
<td>public Integer apply(String s) {</td>
</tr>
<tr>
<td></td>
<td>return s.length();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
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

```java
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());
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[]> 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

```java
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>>);
    TorElse(T);
    TorElseGet(Supplier<T>);
    <X extends Throwable> TorElseThrow(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)

• Challenge: Convert some operation in your Carcassonne solution to use streams... (or at least identify some operations that you might convert)
Stream parallelism: Your mileage may vary

- Consider this for-loop (.96 s runtime; dual-core laptop)
  
  ```java
  long sum = 0;
  for (long j = 0; j < Integer.MAX_VALUE; j++) sum += j;
  ```

- Equivalent stream computation (1.5 s)
  
  ```java
  long sum = LongStream.range(0, Integer.MAX_VALUE).sum();
  ```

- Equivalent parallel computation (.77 s)
  
  ```java
  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 >> 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