Life after Comp215: Java8 Streams, etc.

Dan S. Wallach (Rice University)
Announcements

No labs this week (finish your project!)

Wednesday’s lecture will be a review / Q&A for midterm 2

Friday will be midterm 2 (watch Piazza for room assignments)
We start at 9am sharp. We finish at 9:50am sharp. Don’t be late!
Same “open Internet” exam policies as last time.

See Piazza discussion on due date/late penalty policies for the project
You’ve now outgrown the edu.rice classes!

So you’ll have to learn to work with Java’s “Collections” classes

Before Java8, the Collections classes were all about mutation
But in Java8, they added some nice functional extensions

When you want to get even fancier than that
Google Guava: https://github.com/google/guava
Apache Commons Collections: https://commons.apache.org/proper/commons-collections/
FunctionalJava: http://www.functionaljava.org/

Or there are other programming languages that run on the JVM
Scala, Kotlin, Ceylon, Groovy, Clojure, etc.: all can interoperate with “legacy” Java code
java.util.LinkedList

Constant-time insert to front or end of a list

Also, with Java8, adds some convenience lambdas

```java
import java.util.LinkedList;

public class Test {
    public static void main(String args[]) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Hello"); // goes on the end of the list
        list.add("World"); // goes on the end of the list
        list.addFirst("Rice"); // goes on the front of the list
        list.forEach(System.out::println);
    }
}
```

Prints:
Rice
Hello
World
LinkedList\(<T>\) is also a List\(<T>\)

data java.util.List is a lot like edu.rice.list.IList

General-purpose interface that many different list types implement

Write your code to be reusable, try not to mention the concrete types

```java
public class Test {
    public static void main(String args[]) {
        List<String> list = new LinkedList<>();
        list.add("Hello"); // goes on the end of the list
        list.add("World"); // goes on the end of the list
        list.addFirst("Rice"); // not defined in List! Won't compile!
        list.forEach(System.out::println);
    }
}
```
**LinkedList<T> is also a Deque<T>**

Deque (pronounced “deck”): double-ended queue
Constant-time inserts/queries to the front or the back

**Many variants on this available. Example: LinkedBlockingDeque**
Use for “producer/consumer” communication across threads
(The machinery behind “actor”-style concurrency; learn more in Comp322.)

```java
public class Test {
    public static void main(String args[]) {
        Deque<String> list = new LinkedList<>();
        list.add("Hello"); // goes on the end of the list
        list.add("World"); // goes on the end of the list
        list.addFirst("Rice"); // this one works!
        list.forEach(System.out::println);
    }
}
```
java.util.ArrayList and java.util.Arrays

ArrayList feels like a list, has an array on the inside
You can index into it like an array for O(1) speed (we did this inside BinaryHeap)

Arrays (note: not “Array”) is a utility class to help you deal with arrays
In particular, Arrays.asList() works like our List.of() method

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
What’s missing: `IList.tail()`

Every `IList` has a `head()` and `tail()`, and these are $O(1)$ efficient. Recursive algorithms can process the head, recurse on the tail. There’s no efficient equivalent for any `java.util.List` class!

So what do you do?

`java.util.List` classes define a `clone()` method, but it returns a “shallow” copy. If you make a `clone()`, then mutate, you’re also mutating the parent list — not what you want!

Instead, you can use a “copy constructor”, but it’s $O(n)$.
Taking the tail of a mutating list is $O(n)$

```java
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        Deque<String> tailNames = new LinkedList<>(names);
        tailNames.removeFirst();

        System.out.print("The original is untouched: ");
        names.forEach(System.out::print);
        System.out.print("But the tail is correct: ");
        tailNames.forEach(System.out::print);
        System.out.println("\n");
    }
}
```

Prints:
The original is untouched: AliceBobCharlie
But the tail is correct: BobCharlie
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        Deque<String> tailNames = new LinkedList<>(names);
        tailNames.removeFirst();

        System.out.print("The original is untouched: ");
        names.forEach(System.out::print);
        System.out.print("\nBut the tail is correct: ");
        tailNames.forEach(System.out::print);
        System.out.println("");
    }
}

Prints:
The original is untouched: AliceBobCharlie
But the tail is correct: BobCharlie

Taking the tail of a mutating list is \(O(n)\)

\(O(n)\) deep copy vs. IList’s \(O(1)\) tail
Java8 Streams
Java8 streams

Every collection class now has a .stream() method

```java
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Bob", "Charlie", "Alice");

        String lowerCaseConcat = names.stream()
            .map(String::toLowerCase)
            .sorted()
            .reduce((x, y)->x + " " + y)
            .orElse("Nobody");

        System.out.println(lowerCaseConcat);
    }
}
```

Prints: alice bob charlie
Every collection class now has a `.stream()` method

```java
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Bob", "Charlie", "Alice");

        String lowerCaseConcat = names.stream()
            .map(String::toLowerCase)
            .sorted()
            .reduce((x,y)->x + " " + y)
            .orElse("Nobody");

        System.out.println(lowerCaseConcat);
    }
}
```

Prints: alice bob charlie

Gives us a stream “backed” by the list
Java8 streams

Every collection class now has a `.stream()` method

```java
public class Test {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Bob", "Charlie", "Alice");

        String lowerCaseConcat = names.stream()
            .map(String::toLowerCase)
            .sorted()
            .reduce((x, y) -> x + " " + y)
            .orElse("Nobody");

        System.out.println(lowerCaseConcat);
    }
}
```

Prints: alice bob charlie

All the usual functional operators
Java8 streams

Every collection class now has a .stream() method

```java
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Bob", "Charlie", "Alice");

        String lowerCaseConcat = names.stream()
            .map(String::toLowerCase)
            .sorted()
            .reduce((x,y)->x + " " + y)
            .orElse("Nobody");

        System.out.println(lowerCaseConcat);
    }
}
```

Prints: alice bob charlie

Sorting (w/ optional comparator lambda)
Every collection class now has a `.stream()` method

```java
public class Test {
    public static void main(String args[]) {
        List<String> names = Arrays.asList("Bob", "Charlie", "Alice");

        String lowerCaseConcat = names.stream()
            .map(String::toLowerCase)
            .sorted()
            .reduce((x, y) -> x + " " + y)
            .orElse("Nobody");

        System.out.println(lowerCaseConcat);
    }
}
```

Prints: alice bob charlie

Reduction (foldl), returns Optional
Reduce vs. foldl vs. foldr

Foldl and foldr have well-defined orderings (l-to-r and r-to-l, respectively)
Also a required “accumulator”; even empty lists can be folded

Stream.reduce can happen in parallel, ordering is (optionally) undefined
Under the hood, some streams try to maintain their ordering, others don’t

Stream.reduce has extra features for running in parallel

```
<U> U reduce(U identity, 
    BiFunction<U,? super T,U> accumulator, 
    BinaryOperator<U> combiner)
```

“accumulator” function is the exact same lambda we use for foldl
“combiner” function merges partial accumulations (computed in parallel)
Some simple examples

Let’s say we have a List<String>....

How many strings have the name “Dan” in them?
```java
int result = list.stream()
    .map(x->x.contains("Dan")?1:0)
    .reduce((x,y)->x+y)
    .orElse(0);
```

How long is the longest string?
```java
int longest = list.stream()
    .map(x->x.length())
    .reduce((x,y)->(x>y)?x:y)
    .orElse(0);
```
Making the simple, simpler

Let’s say we have a List<String>....

*How many strings have the name “Dan” in them?*
int result = list.stream()
  .filter(x->x.contains("Dan"))
  .count();

*How long is the longest string?*
int longest = list.stream()
  .map(x->x.length())
  .max(Integer::compare)
  .orElse(0);
Convenience: Stream.of

Before:
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()...

After:
Stream.of("Alice", "Bob", "Charlie")...
Infinite streams, just like our LazyList

How about iterating a function over and over again?

```java
Stream<Integer> evenNumbers = Stream.iterate(0, x->x+2);
```

Lazy evaluation: it doesn't actually compute those values until asked. But we can't just do `reduce()` on this. It would run forever.

```java
evenNumbers.limit(100).max(Integer::compare).orElse(0) → 198
```

If you've got some arbitrary function that supplies different values

```java
Stream<Integer> magicNumbers = Stream.generate(()->getNextNum());
```
Infinite stream aren’t exactly lazy lists

Lazy lists act just like regular lists
You can take the `tail()`, etc. All methods on `edu.rice.list.IList` work on `LazyList`.

Java8 Streams are not lists, they’re ephemeral
You ultimately ask for some result, and then all the intermediate stuff is gone.
You have to convert from streams back to regular lists again.
Getting results out of a stream

So far, we’ve shown you reduction operators (reduce, max, length, ...)
But you can also “collect” back into a list again
List<Whatever> list = stream.collect(Collectors.toList());

Many different kinds of “collectors” available
String result = stream.collect(Collectors.joining("", ""));

Equivalent to IList.join()
Files → Streams

java.io.BufferedReader.lines() returns a stream of strings

java.nio.Files has several useful methods
  .find() lets you search for files whose names match any predicate (e.g., *.txt)
  .list() gives you a stream of file Path objects
  .lines() goes from a Path to a stream

Example: Print all the file names in the current directory:
try {
  Files.list(new File(".").toPath())
    .forEach(System.out::println);
} catch (IOException e) {
  System.out.println("Unexpected failure: " + e.toString());
}
Files → Streams

`java.io.BufferedReader.lines()` returns a stream of strings

`java.nio.Files` has several useful methods
- `.find()` lets you search for files whose names match any predicate (e.g., `*.txt`)
- `.list()` gives you a stream of file `Path` objects
- `.lines()` goes from a `Path` to a stream

Example: Print all the file names in the current directory:
```java
try {
    Files.list(new File(".").toPath())
        .forEach(System.out::println);
} catch (IOException e) {
    System.out.println("Unexpected failure: " + e.toString());
}
```
Concatenating streams

Non-trivial example: every file in the current directory to a single stream

```java
try {
    Files.list(new File(".").toPath())
        .flatMap(path -> {
            Stream<String> header = Stream.of("==============" + path.toString() + "==========");
            try {
                if(path.toFile().isFile())
                    return Stream.concat(header, Files.lines(path));
                else
                    return Stream.concat(header, Stream.of("not a file"));
            } catch (IOException e) {
                // couldn't open the file, so return the exception as a string in a stream
                return Stream.concat(header, Stream.of(e.toString()));
            }
        })
        .forEach(System.out::println);
} catch (IOException e) {
    System.out.println("Unexpected error: " + e.toString());
}
```
Concatenating streams

Non-trivial example: every file in the current directory to a single stream

```
try {
    Files.list(new File(".").toPath())
        .flatMap(path -> {
            Stream<String> header = Stream.of("==============" + path.toString() + "==========");
            try {
                if(path.toFile().isFile())
                    return Stream.concat(header, Files.lines(path));
                else
                    return Stream.concat(header, Stream.of("not a file"));
            } catch (IOException e) {
                // couldn't open the file, so return the exception as a string in a stream
                return Stream.concat(header, Stream.of(e.toString()));
            }
        })
        .forEach(System.out::println);
} catch (IOException e) {
    System.out.println("Unexpected error: " + e.toString());
}
```
Word frequency count

How many times does each word in a stream occur?

```java
Map<String, Integer> wc = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
        x -> 1,
        (x,y) -> x+y));

wc.forEach((key,value)->
    System.out.println(key + "=" + value));

Dorothy=1
Bob=1
Eve=1
Alice=3
Charlie=1
```
Word frequency count

How many times does each word in a stream occur?

Map<String, Integer> wc = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
        x -> 1,
        (x,y) -> x+y));

wc.forEach((key, value) ->
    System.out.println(key + " = " + value));

Dorothy=1
Bob=1
Eve=1
Alice=3
Charlie=1
Digression: What’s a Map?

java.util.Map: similar to our IMap interface
HashMap: backed by a hash table
TreeMap: backed by a fancy red-black tree (faster than our treap)
Map.Entry: similar to our KeyValue class

Much like edu.rice.list.IList vs. java.util.List, these are mutating interfaces
Can I turn a Map into a Stream?

Yes!  
\[
\text{Map} \rightarrow \text{Set\langle Map.Entry<K,V\rangle} \rightarrow \text{Stream\langle Map.Entry<K,V\rangle}
\]

Map\langle String, Integer \rangle wc = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
  .collect(Collectors.toMap(x -> x,
                          x -> 1,
                          (x,y) -> x+y));

wc.entrySet().stream()
    .filter(entry->entry.getValue() > 1)
    .forEach(System.out::println);

Alice=3
Can I turn a Map into a Stream?


Map<String, Integer> wc = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                                x -> 1,
                                (x,y) -> x+y));

wc.entrySet().stream()
    .filter(entry->entry.getValue() > 1)
    .forEach(System.out::println);

Alice=3
Easy to go from one to the other and back again.

```java
Map<String, Integer> wc = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                             x -> 1,
                             (x, y) -> x+y));

Map<String, Integer> bigWords =
    wc.entrySet()
    .stream()
    .filter(entry->entry.getValue()>1)
    .collect(Collectors.toMap(entry->entry.getKey(),
                               entry->entry.getValue()));

System.out.println(bigWords);

{Alice=3}
```

Ugly, but functional

Exception if two entries with same key, optional lambda argument merges values when keys equal
Putting it all together in one pipeline:

Map<String, Integer> bigWords =
Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                                x -> 1,
                                (x, y) -> x+y))
    .entrySet()
    .stream()
    .filter(entry -> entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey,
                                Map.Entry::getValue));

System.out.println(bigWords);

{Alice=3}
Putting it all together in one pipeline:

```java
Map<String,Integer> bigWords = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                                x -> 1,
                                (x,y) -> x+y))
    .entrySet()
    .stream()
    .filter(entry->entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey,
                               Map.Entry::getValue));

System.out.println(bigWords);

{Alice=3}
```
Putting it all together in one pipeline:

```java
Map<String, Integer> bigWords = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                              x -> 1,
                              (x, y) -> x + y))
    .entrySet()
    .stream()
    .filter(entry -> entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey,
                                Map.Entry::getValue));
System.out.println(bigWords);

{Alice=3}
```
Putting it all together in one pipeline:

```java
Map<String, Integer> bigWords = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x, 
        x -> 1,
        (x,y) -> x+y))
    .entrySet()
    .stream()
    .filter(entry->entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey, 
        Map.Entry::getValue));

System.out.println(bigWords);

{Alice=3}
```
Putting it all together in one pipeline:

```java
Map<String, Integer> bigWords = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
        x -> 1,
        (x,y) -> x+y))
    .entrySet()
    .stream()
    .filter(entry->entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey, 
        Map.Entry::getValue));

System.out.println(bigWords);

{Alice=3}
```
Putting it all together in one pipeline:

```java
Map<String, Integer> bigWords = Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
   .collect(Collectors.toMap(x -> x,
                                x -> 1,
                                (x, y) -> x+y))
   .entrySet()
   .stream()
   .filter(entry -> entry.getValue() > 1)
   .collect(Collectors.toMap(Map.Entry::getKey,
                            Map.Entry::getValue));

System.out.println(bigWords);

{Alice=3}
```
Putting it all together in one pipeline:

```java
Map<String, Integer> bigWords =
Stream.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
    .collect(Collectors.toMap(x -> x,
                               x -> 1,
                               (x, y) -> x+y))
    .entrySet()
    .stream()
    .filter(entry -> entry.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey,
                               Map.Entry::getValue));
```

```
System.out.println(bigWords);

{Alice=3}
```
IMap<String, Integer> bigWords2 =
   TreeMap.fromList(
      TreeMap.fromList(
         LazyList.of("Alice", "Bob", "Charlie", "Dorothy", "Eve", "Alice", "Alice")
            .map(string -> new KeyValue<>(string, 1)),
            (val1, val2) -> val1 + val2)
         .toList()
         .filter(kv -> kv.getValue() > 1));

Not all that different (and a bit annoying because it doesn’t read top to bottom)
Parallelism!

Comp215 isn't about parallelism. That's Comp322. However...

Collection classes have a `.stream()` method
Generates a stream, like we know and love

They also have a `.parallelStream()` method
You get to use all the same stream operations as before

Multicore/multithreaded speedup with (almost) zero coding effort!
Even the `.sorting()` function runs in parallel
Collectors.toMap() becomes Collectors.toConcurrentMap()
Warnings and caveats

Ordering isn't guaranteed
Original order is (possibly) lost to allow for parallelism
Some stream variants preserve ordering
  Or just sort on the way out
  Or just store the result in a set or map

Use lambdas that are stateless since you have no idea when they’ll run
Some web pages that teach how streams work get this wrong
(Remember our advice on StackOverflow: don’t trust everything you read...)

Once you understand streams, you’re ready for MapReduce / Hadoop / etc.
Parallel maps, spread across clusters of thousands of machines
This week: check out edu.rice.stream

We’ve given you adapters from IList to Stream
Make it easy for you to gain parallelism speedups

IList to Stream conversion?
Version 1: converts IList to an Iterator, then Iterator to a Stream
Version 2: copies everything from IList to ArrayList

Stream to IList conversion?
Converts from Stream to a List (we don’t know or care what kind of list)
LazyList.ogenerate() then reads out the contents of that list
Parallel speedup tests! Number of available CPUs: 12 (6 core MacPro)

============= List vs. Stream performance (listLength = 100000, hashRepeats = 1) ===============

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per hash</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular IList</td>
<td>4.653 μs</td>
<td></td>
</tr>
<tr>
<td>sequential stream</td>
<td>2.246 μs</td>
<td></td>
</tr>
<tr>
<td>parallel stream</td>
<td>2.226 μs</td>
<td></td>
</tr>
<tr>
<td>par-array stream</td>
<td>2.220 μs</td>
<td></td>
</tr>
</tbody>
</table>

PARALLEL STREAM SPEEDUP: 1.009x
PAR-ARRAY STREAM SPEEDUP: 1.011x

== List vs. Stream performance (listLength = 100000, hashRepeats = 10) ===============

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per hash</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular IList</td>
<td>6.550 μs</td>
<td></td>
</tr>
<tr>
<td>sequential stream</td>
<td>5.347 μs</td>
<td></td>
</tr>
<tr>
<td>parallel stream</td>
<td>1.975 μs</td>
<td></td>
</tr>
<tr>
<td>par-array stream</td>
<td>2.578 μs</td>
<td></td>
</tr>
</tbody>
</table>

PARALLEL STREAM SPEEDUP: 2.707x
PAR-ARRAY STREAM SPEEDUP: 2.074x

== List vs. Stream performance (listLength = 10000, hashRepeats = 100) ===============

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per hash</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular IList</td>
<td>45.112 μs</td>
<td></td>
</tr>
<tr>
<td>sequential stream</td>
<td>44.577 μs</td>
<td></td>
</tr>
<tr>
<td>parallel stream</td>
<td>18.414 μs</td>
<td></td>
</tr>
<tr>
<td>par-array stream</td>
<td>8.623 μs</td>
<td></td>
</tr>
</tbody>
</table>

PARALLEL STREAM SPEEDUP: 2.421x
PAR-ARRAY STREAM SPEEDUP: 5.170x

== List vs. Stream performance (listLength = 1000, hashRepeats = 1000) ===============

<table>
<thead>
<tr>
<th>Method</th>
<th>Time per hash</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular IList</td>
<td>434.842 μs</td>
<td></td>
</tr>
<tr>
<td>sequential stream</td>
<td>460.512 μs</td>
<td></td>
</tr>
<tr>
<td>parallel stream</td>
<td>70.971 μs</td>
<td></td>
</tr>
<tr>
<td>par-array stream</td>
<td>70.971 μs</td>
<td></td>
</tr>
</tbody>
</table>

PARALLEL STREAM SPEEDUP: 1.115x
PAR-ARRAY STREAM SPEEDUP: 6.489x
Once the per-task cost is big enough, parallelism pays off
And the code is nearly identical

Ordinary IList

```csharp
IList<KeyValue<Integer, Long>> hashedValues = IList.mapkv(slowFunction, numberList);
IMap<Integer, Long> result1 = TreapMap.fromList(hashedValues);
```

Parallel stream

```csharp
IList<KeyValue<Integer, Long>> hashedValuesParallelStream = streamToList(
    listToStream(numberList, true).map(val -> new KeyValue<>(val, slowFunction.apply(val))));
IMap<Integer, Long> result3 = TreapMap.fromList(hashedValuesParallelStream);
```
And the code is nearly identical

**Ordinary IList**

```csharp
IList<KeyValue<Integer, Long>> hashedValues = IList.mapkv(slowFunction, numberList);
IMap<Integer, Long> result1 = TreapMap.fromList(hashedValues);
```

**Parallel stream**

```csharp
IList<KeyValue<Integer, Long>> hashedValuesParallelStream = streamToList(
    listToStream(numberList, true).map(val -> new KeyValue<>(val, slowFunction.apply(val)));
IMap<Integer, Long> result3 = TreapMap.fromList(hashedValuesParallelStream);
```

Convert from an IList to a parallel stream
And the code is nearly identical

Ordinary IList

```csharp
IList<KeyValue<Integer, Long>> hashedValues = IList.mapkv(slowFunction, numberList);
IMap<Integer, Long> result1 = TreapMap.fromList(hashedValues);
```

Parallel stream

```csharp
IList<KeyValue<Integer, Long>> hashedValuesParallelStream = streamToList(
    listToStream(numberList, true).map(val -> new KeyValue<>((val, slowFunction.apply(val))));
IMap<Integer, Long> result3 = TreapMap.fromList(hashedValuesParallelStream);
```
And the code is nearly identical

Ordinary IList

```csharp
IList<KeyValue<Integer, Long>> hashedValues = IList.mapkv(slowFunction, numberList);
IMap<Integer, Long> result1 = TreapMap.fromList(hashedValues);
```

Parallel stream

```csharp
IList<KeyValue<Integer, Long>> hashedValuesParallelStream = streamToList(
    listToStream(numberList, true).map(val -> new KeyValue<>(val, slowFunction.apply(val))));
IMap<Integer, Long> result3 = TreapMap.fromList(hashedValuesParallelStream);
```

Convert back to a list (lazily)
And the code is nearly identical

**Ordinary IList**

```csharp
IList<KeyValue<Integer, Long>> hashedValues = IList.mapkv(slowFunction, numberList);
IMap<Integer, Long> result1 = TreapMap.fromList(hashedValues);
```

**Parallel stream**

```csharp
IList<KeyValue<Integer, Long>> hashedValuesParallelStream = streamToList(
    listToStream(numberList, true).map(val -> new KeyValue<>(val, slowFunction.apply(val)));
IMap<Integer, Long> result3 = TreapMap.fromList(hashedValuesParallelStream);
```

Load back into a TreapMap
Other things you’ve learned carry on

JetBrains IntelliJ IDEA
Widely used in industry, also the basis for Android Studio
Supports many languages besides just Java

SparkJava web server, j2html “fluent” builders, JUnit / Mockito testing
You should teach yourself to use the Maven or Gradle build systems

You won’t use our JSON library, but there are tons of alternatives
See also our discussion on using protocol buffers
Java serialization / Java RMI: just say no!

Our logging library is a direct descendant of Android’s logging
Other Java logging tends to get much more complicated
Consider moving on from Java

Java’s doing well for a 20-year-old programming language, but...

Libraries are old and hacked up (versus being built for functional behavior)

Limited ways to externally add a method to a class (default methods on interfaces)
   Leads to Utils/Adapters classes with only static methods; resulting code doesn’t read top to bottom

Null pointers are the “billion dollar mistake”

Generic type signatures can fill the screen, and you type them again and again
   But they help you catch tons of bugs!

“Design patterns” have led to amazingly ugly, unreadable code

Alternately, in Scala/Kotlin/Ceylon/etc.

Easy to define “extension” methods on a class
   Example: listToStream and streamToList could then appear in line, top to bottom

Type systems that make it easier to work safely with null references

Type inference in more places, (val x = ...), so less declaring types
Some Rice classes that build on Comp215

**Comp311: Functional programming (every fall)**
- Beautiful “functional data structures” that rely on laziness for high performance
- Functional-reactive programming (like streams, but dealing with events that haven’t happened yet)

**Comp321: Introduction to computer systems (every spring)**
- Teaches you the C programming language, learn more about the machinery that make Java possible

**Comp322: Fundamentals of parallel programming (every spring)**
- Asynchronous computation, futures, actors, parallelism, ...
- Will take advantage of the functional thinking you’ve learned here

**Comp411: Principles of programming languages (every spring)**
- Lots of depth where we brushed over topics (e.g., lexical scope, polymorphism, type inference)

**Comp412: Compiler construction (every fall)**
- More fun with parsing and grammars (never mind code generation and optimization)

**Comp440: Artificial intelligence (every fall)**
- The genetic algorithms you saw here are the tip of a very deep iceberg
So try to avoid mutation...
It’s worth it.