159.341 Programming Languages, Algorithms & Concurrency

Java Concurrency - Part 3

Daniel Playne
d.p.playne@massey.ac.nz

Java Streams

Java Streams were introduced in Java 8 as an efficient way to carry out operations on data.

Java streams support *functional* style operations on *streams* of elements.

Streams are not really data structures themselves (nor do they have anything to do with Java I/O streams).

Java Streams

Streams are different from collections in a number of ways:

- A stream does not store data.
- *Functional* - an operation on the stream produces a result but does not modify the source.
- *Lazy* - many stream operations are implemented lazily.
- *Possibly unbounded* - streams may have an infinite size.
- *Consumable* - elements of a stream are visited only once.

Java Streams

A ***Stream*** stream is a sequence of elements but rather than storing elements like a data structure, they carry values through a pipeline of operations.

Aggregate operations are operations that process elements from a Java ***stream*** rather than directly from a collection.

Java Streams

If we wanted to go through a list of integers and print out only the even numbers, we might write code something like the following:

```
List<Integer> numbers = new ArrayList<Integer>();
...

for(Integer i : numbers) {
   if(i % 2 == 0) {
      System.out.println(i);
   }
}
```

Java Streams

To write similar functionality using Java streams we could instead write the following:

```
numbers.stream()
       .filter(i -> i % 2 == 0)
       .forEach(i -> System.out.println(i));
```

Note the use of Java lambda functions in this implementation.

Java Streams

This is an example of a *stream pipeline*.

First a stream is created from a list of integers

```
numbers.stream()
```

The stream does not store anything itself, instead it can be thought of as a way of accessing a sequence of elements.

Java Streams

The next operation

```
.filter(i -> i % 2 == 0)
```

Will *produce* another stream which consists of only the even integers in the original stream.

Again, this stream does not store another copy of these elements. When an element from the stream is accessed, only the even numbers will make it through this operation.

Java Streams

The final operation

```
.forEach(i -> System.out.println(i))
```

Is a *terminal operation* that will consume the elements from the previous stream.

In this case it just prints them to the output.

Java Streams

Each stream computation has a common structure:

- A ***stream source***
- Zero or more ***intermediate operations***
- A single ***terminal operation***.

A *stream source* is the initial series of elements in the stream pipeline, this will often be a Java collection but there are a variety of possible stream sources.

Java Streams

Stream sources:

- `Stream.empty()` - stream with no elements
- `Collection.stream()` - stream from a collection
- `Stream.of(T[])` - stream from an array
- `BufferReader.lines()` - stream from a `BufferedReader`
- a generator function
- etc.

Java Streams

Intermediate operations are operations that transform a stream into another stream.

Intermediate operations may transform elements, filter out elements, rearrange their order in the stream etc.

- `filter(Predicate<T>)` - filter elements by a predicate
- `map(Function<T,U>)` - apply function to each element
- `sorted(Comparator<T>)` - sort elements by Comparator
- etc.

Java Streams

Intermediate operations in Java stream are *lazy*.

Defining an *intermediate operation* on a stream simply describes the pipeline but does not actually require any computation to be performed.

A distinction can also be made between *stateless* and *stateful* intermediate operations.

Java Streams

Operations such as `filter` or `map` are *stateless* as they can be applied to each element in the list independently.

stateful operations such as `sorted` or `distinct` cannot be applied independently - they require information about the other elements in the stream.

Java Streams

Finally, each Java stream pipeline ends with a *terminal* operation which produces a side-effect or a result.

The rest of the stream pipeline only needs to be computed when a terminal operation is executed.

Java Streams

Examples of *terminal operations* include:

- `forEach(Consumer<T>)` - apply an action to each element
- `reduce(...)` - reduce elements in the stream to a single value
- `toArray()` - build an array from the elements in the stream
- `collect()` - build a collection from the elements in the stream
- `count()` - count the number of elements in the stream
- `min(Comparator<T>)` - find the minimal element in the stream
- `max(Comparator<T>)` - find the maximal element in the stream

Java Streams

Java streams describe computation rather than data.

The nature of Java streams allows lazy evaluation and operation fusion across the pipeline.

Another advantage is that they can be easily converted to be executed in parallel.

Java Streams

The functional style of Java streams mean there are opportunities for *implicit parallelism*.

Because Java streams are essentially a functional pipeline, parallelising them can be performed far more simply than an arbitrary Java program.

In many cases this can be achieved by simply changing `stream()` to `parallelStream()`.

Java Streams

For example:

```
List<Integer> number = new ArrayList<Integer>();
for(int i = 0; i < 1000000; i++) {
   numbers.add((int)(Math.random()*1000));
}

numbers.stream().reduce(0, (a,b) -> a + b);
numbers.parallelStream().reduce(0, (a,b) -> a + b);
```

Java Streams

The advantages of using Java stream also come with many of the restrictions of functional languages.

Since Java is a fundamentally imperative programming language, care must be taken with expressions that may have side-effects.

Java Streams

For example:

```
List<Integer> numbers = new ArrayList<Integer>();
for(int i = 0; i < 1000000; i++) {
   numbers.add((int)(Math.random()*1000));
}

numbers.stream()
       .forEach(i -> {
          numbers.add(i);
       });
```

Produces:

```
Exception in "main"
java.util.ConcurrentModificationException
```

Java Streams

Most stream operations are required to be *non-interfering* and *stateless*.

Non-interfering operations may not modify the stream source.

Stateless operations do not access any state that may be modified during the lifetime of the operation.

Java Streams

For example:

```
int total = 0;

numbers.stream()
       .filter(i -> i % 2 == 0)
       .forEach(i -> total = total + i);
```

Produces a compiler error:

```
error: local variables referenced from a lambda expression must be final or effectively
final
```

Java Streams

Unfortunately this can be bypassed by wrapping the primitive in another object (even just an array).

```
Integer total[] = new Integer[1];

numbers.stream()
       .filter(i -> i % 2 == 0)
       .forEach(i -> total[0] += i);
```

Which is a potentially dangerous thing to do.

Java Streams

Trying to replace this with a parallel stream will produce incorrect results.

```
Integer total[] = new Integer[1];
total[0] = 0;

numbers.parallelStream()
       .filter(i -> i % 2 == 0)
       .forEach(i -> total[0] += i);
```

Obviously there is now a race condition in the `forEach` operation.

Java Streams

This could be addressed by synchronising the critical section in the `forEach` operation.

```
Integer total[] = new Integer[1];
total[0] = 0;

numbers.parallelStream()
       .filter(i -> i % 2 == 0)
       .forEach(i -> {synchronized(total) {total[0] += i;}});
```

But a much better solution would be to properly use a `reduce` operation.

Summary

Java streams provide a useful way of describing a functional-style computation pipeline.

The advantages to this approach include opportunities for implicit parallelism and other optimisations (lazy evaluation etc).

A good understanding of functional programming and concurrency helps to understand how to utilise streams (and the restrictions on them).