COMP8710 Advanced Java for Programmers

Lecture 15
Parallel stream & JavaFX Introduction (1)

Yang He

Topics

- Introduction to Java Streams
- Stream operations
- Collections vs. streams
- Matching with Optional
- Stateless vs. stateful
- Infinite streams
- Parallel streams
- JavaFX Introduction

Parallel streams

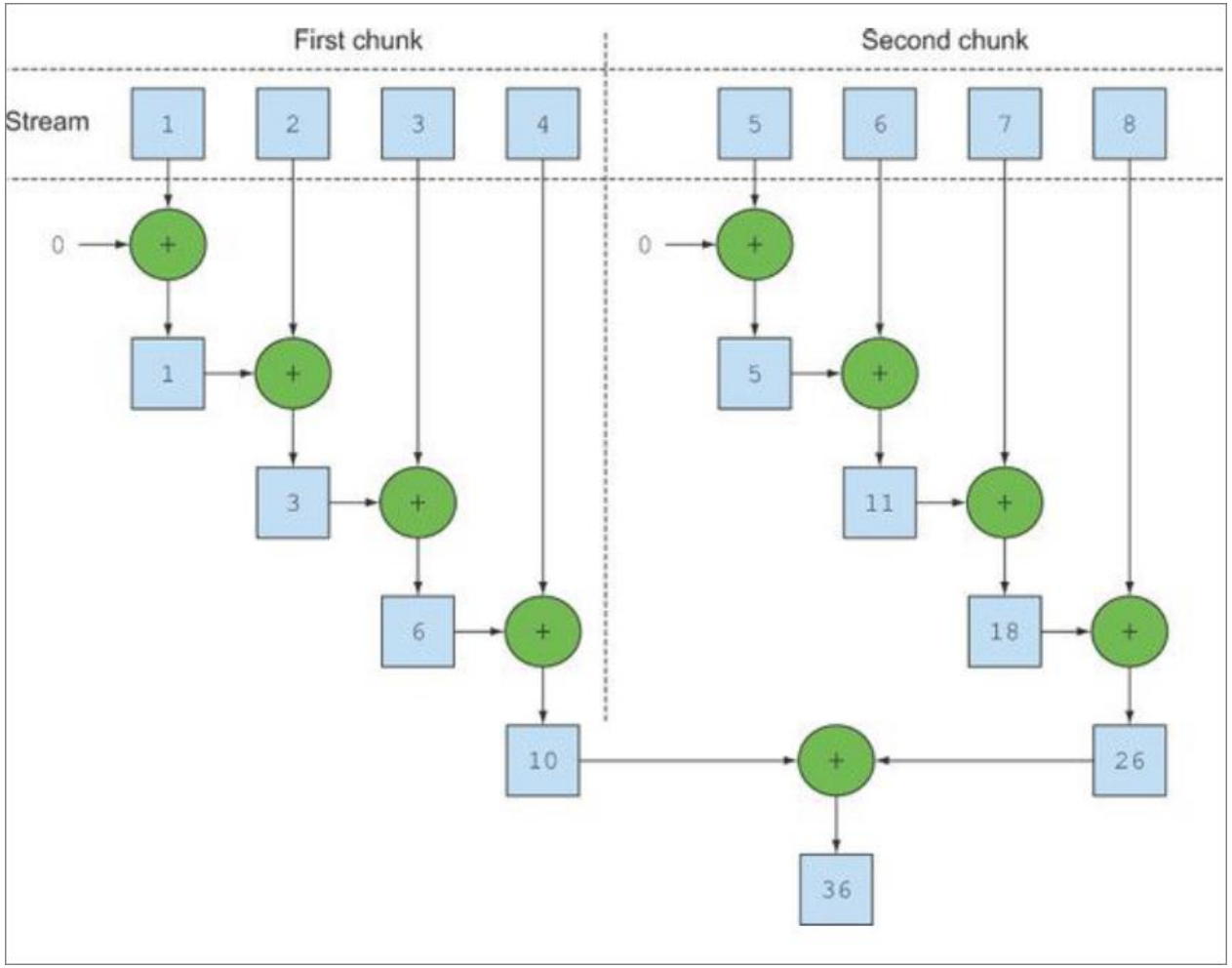
- Parallelisation with streams is easy . . .
- E.g. Define a method that takes a number n and returns the sum of the first n natural numbers (i.e. from 1 to n)
  - The sequential function:

```
public static long sequentialSum(long n) {
   return Stream.iterate(1L, i -> i + 1)
                .limit(n)
                .reduce(0L, Long::sum);
}
```

  - becomes parallel:

```
public static long parallellSum(long n) {
   return Stream.iterate(1L, i -> i + 1)
                .limit(n)
                .parallel()
                .reduce(0L, Long::sum);
}
```

- E.g. 2 cores/processors



What parallel does (1)

- Calling the method `parallel` on a sequential stream
  - Does not imply any concrete transformation on the stream itself
  - It only signals that you want to parallelise the operations that follow
- When the method `parallel` is invoked:
  - the Stream is *internally* divided into multiple chunks
  - the reduction operation can work on the various chunks *independently* and in parallel
  - the same reduction operation combines the values resulting from the partial reductions of each sub-stream
  - producing the result of the reduction process on the whole initial stream

What parallel does (2)

- Parallel streams internally use the default ForkJoinPool
- It runs as many threads as you have processors/cores
- You can override this, but not recommended!

Good parallelisation

- It is easy to transform a sequential stream into a parallel one, but it is not so straightforward to make parallelisation:
  - Correct
    - The value obtained from the parallel stream is the same as the sequential one
  - Efficient
    - The computation of parallel stream is faster than the sequential one

Of course, correctness comes before performance!

Correct parallelisation: counter-example

- Assume that we want to compute the sum of the first n integers, using an accumulator as shown on the right
- Is this a good implementation?

No, the problem is that the method invoked inside forEach block has the side effect of changing the mutable state of an object shared among the multiple threads.

```
public static class Accumulator
{
        private long total = 0;
        public void add (long value) {
                total += value;
        }
         public long getTotal() { return total; }
}

public static long accumParallelSum(long n) {
        var accumulator = new Accumulator();
        LongStream.rangeClosed(1, n)
                          .parallel()
                          .forEach(accumulator::add);
        return accumulator.getTotal();
}
```

Stream: parallel processing

- Streams naturally enable parallelisation on certain tasks
- Can we execute these operations in parallel?

```
Stream.iterate(1, i -> i + 1)
      .limit(10)
      .map(j -> j * 2)
      .forEach(System.out::println);
```

```
Stream.iterate(1L, i -> i + 1)
      .limit((long) Math.pow(10,7))
      .reduce(0L, Long::sum);
```

The `iterate` operation is hard to split into chunks that can be executed independently because the input of one function application always depends on the result of the previous application.

Demo:
Measuring performances

`ParellelStream.java`

Stream flags (1)

- Streams have several characteristics which may speed some operations:
  - SIZED: size is known.
  - DISTINCT: elements are pairwise distinct wrt. equals for objects (and == for primitive types)
  - SORTED: elements are sorted (according to their natural) ordering, see Comparator)
  - ORDERED: the stream has a “meaningful encounter order” (e.g. lists are ordered, but sets and maps are not).

Stream flags (2)

- Each operation may affect the characteristics, e.g.,
  - `map` preserves SIZED and ORDERED but not DISTINCT nor SORTED
  - `sorted` preserves SIZED and DISTINCT, and adds SORTED

Effective parallel streams (1)

- Some consideration:
  - If in doubt, measure.
    - A parallel stream isn't always faster, especially for small amounts of data
  - Watch out for (automatic) boxing
    - E.g. use `LongStream.rangeClosed` instead of a stream of `List<Long>`
  - Some operations naturally perform worse on a parallel stream than on a sequential stream, e.g. `limit` and `findFirst`. You can use `unordered()`
  - Consider the total computational cost of the pipeline:
    no. of elements x cost of processing one element

Effective parallel streams (2)

- Take into account how well the data structure underlying the stream decomposes (e.g. ArrayList vs. LinkedList)
- Beware that some intermediate operations in the pipeline may modify the stream and thus change the performance of the decomposition process
  - e.g. the size of a stream may change after a filter operation
- Consider the cost of combining intermediate result

- If you need to fine-tune the parallelisation, have a look at the Fork/Join framework and the `Spliterator` interface (*out of the scope of this module*)

Parallel-friendliness of some stream sources

Source	Decomposability
ArrayList	Excellent
LinkedList	Poor
IntStream.range	Excellent
Stream.iterate	Poor
HashSet	Good
TreeSet	Good

Benefits of Java Streams

- Concise and readable
  - Declarative, focusing on what functions perform, not how to perform
- Flexible and composable
  - Functions are automatically connected, i.e. chained
- Simplified scalability
  - Parallelise performance - no need to write any multi-thread code

Exercises

- Identify errors:

1) `menu.stream().map(Dish::type);`

Missing a terminal operation.

2)
```
Stream<Dish> s = menu.stream();
s.filter(d -> d.calories() < 300).findAny();
s.forEach(System.out::println);
```

A stream can be traversed only once!

3) `menu.stream().peek(menu::remove).forEach(System.out::println)`

The collection of the stream has been modified.

JavaFX Introduction

GUI in Java – History

- Abstract Window Toolkit (AWT) — 1995
- Swing — 1998
- JavaFX —2008, modern GUI toolkit
  - Replaces AWT & Swing
  - Part of JRE/JDK from v7 (2012) till v11 (2018)
  - Support and development is now separately from the JDK
- Now managed by the OpenJFX project
  - Download JavaFX libraries at: https://gluonhq.com/products/javafx/

What is JavaFX?

- It is a Java library that is used to develop
  - Desktop applications
  - Rich Internet Applications (RIA)
- JavaFX applications can run on multiple platforms including
  - Desktops (e.g. Windows, Linux and Mac OS)
  - Web
  - Mobile

Where is JavaFX used?

- The main domain of application of JavaFX is centred in business applications, which tend to be:
  - Big: they have many screens
  - Complex: a lot of rules drive the application processing
  - Used by employees: the main users of the application are employees who interact with it as part of their daily work
  - Long-term-oriented: the lifecycle of an application is long (10 years)

Source: https://www.oracle.com/technical-resources/articles/java/casa.html

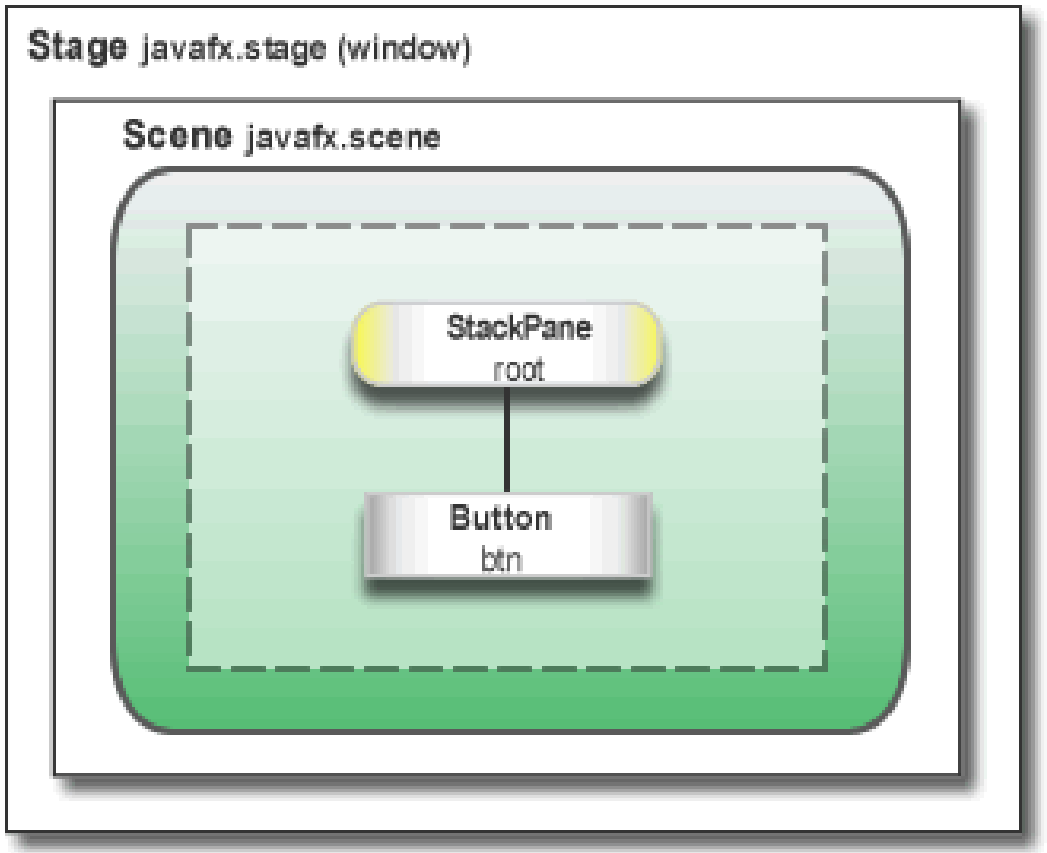
JavaFX modules

- JavaFX 21 API https://openjfx.io/javadoc/21/

Modules

Module	Description
javafx.base	Defines the base APIs for the JavaFX UI toolkit, including APIs for bindings, properties, collections, and events.
javafx.controls	Defines the UI controls, charts, and skins that are available for the JavaFX UI toolkit.
javafx.fxml	Defines the FXML APIs for the JavaFX UI toolkit.
javafx.graphics	Defines the core scenegraph APIs for the JavaFX UI toolkit (such as layout containers, application lifecycle, shapes, transformations, canvas, input, painting, image handling, and effects), as well as APIs for animation, css, concurrency, geometry, printing, and windowing.
javafx.media	Defines APIs for playback of media and audio content, as part of the JavaFX UI toolkit, including `MediaView` and `MediaPlayer`.
javafx.swing	Defines APIs for the JavaFX / Swing interop support included with the JavaFX UI toolkit, including `SwingNode` (for embedding Swing inside a JavaFX application) and `JFXPanel` (for embedding JavaFX inside a Swing application).
javafx.web	Defines APIs for the WebView functionality contained within the the JavaFX UI toolkit.

The theatre metaphor



- Stage: window
- Scene: window content
- (Stack) Pane: layout manager
- Button: UI controls

JavaFX applications: basics

- The main class for a JavaFX application
  - extends `javafx.application.Application`
  - overrides the `start` method which is automatically called when the application is launched, i.e. calling the method `launch`, from within the main method
- Note:
  - A `Stage` object is essentially a window
  - A primary Stage is automatically created by the JVM when the application is launched
  - You can create additional Stage objects if you want to open additional windows