CS2030 Lecture 7

Declarative Programming with Integer Streams

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2018 / 2019

Lecture Outline

- Declarative versus imperative programming
- Internal versus external iteration
- Stream programming concepts using `java.util.stream.IntStream`
  - Stream elements
  - Stream pipelines
  - Intermediate and terminal operations
  - Lazy and eager evaluations
  - Lambda expressions
  - Mapping
  - Reduction
  - Method references
  - Infinite streams

External Iteration

- An imperative loop that specifies *how* to loop and sum

```
int sum = 0;
for (int x = 1; x <= 10; x++) {
    sum += x;
}
```

- Realize the variables `i` and `sum` *mutates* at each iteration
- Errors could be introduced when
  - `sum` is initialized wrongly before the loop
  - looping variable `i` is initialized wrongly
  - loop condition is wrong
  - increment of `i` is wrong
  - aggregation of `sum` is wrong

Internal Iteration

- A *declarative* approach that specifies *what* to do

```
int sum = IntStream
    .rangeClosed(1, 10)
    .sum();
```

- `sum` is assigned with the result of a **stream pipeline**
- Literal meaning "for the range 1 through 10, sum them"
- A **stream** is a sequence of elements on which tasks are performed; the stream pipeline moves the stream's elements through a sequence of tasks
- No need to specify how to iterate through elements or use any *mutatable* variables — no variable state, no problem 😄
- `IntStream` handles all the iteration details
- A key aspect of functional programming

Streams and Pipelines

- A stream pipeline starts with a **data source**
- Static method `IntStream.rangeClosed(1, 10)` creates an `IntStream` containing the ordered sequence $1, 2, \ldots, 9, 10$
  - `range(1, 10)`produces the ordered sequence $1, 2, \ldots, 8, 9$
- Instance method `sum` is the processing step, or **reduction**
  - it reduces the stream of values into a single value
  - Other reductions include `count`, `min`, `max`, `average`

```
long count = IntStream
    .rangeClosed(1, 10)
    .count();
```

```
int max = IntStream
    .rangeClosed(1, 10)
    .max()
    .getAsInt();
```

- Reductions are **terminal operations** that initiate a stream pipeline's processing so as to produce a result

Mapping

- Most stream pipelines contain **intermediate operations** that specify tasks to perform on a stream's elements before a terminal operation produces a result
- **Mapping** is a common intermediate operation is which
  - transforms a stream's elements to new values
  - resulting stream has the same number of elements
  - type of the mapped elements can be different from that of the original stream's elements
- Example, given the following external iteration

```
int sum = 0;
for (int x = 1; x <= 10; x++) {
    sum += (2 * x);
}
```


Mapping

- Using internal iteration

```
int sum = IntStream
    .rangeClosed(1, 10)
    .map(/* some mapping operation */)
    .sum();
```

- `map` is the processing step that would map each element in the stream to that multiplied by 2, giving a stream of even integers
- From Java 9 API,

```
IntStream map(IntUnaryOperator mapper)

Returns a stream consisting of the results of applying
the given function to the elements of this stream.

This is an intermediate operation.
```


`IntStreams`'s `map`

- `map` operation takes in an instance of a `IntUnaryOperator` as argument
- `IntUnaryOperator` is a **functional interface** with a *single abstract method*

```
int applyAsInt(int operand)

Applies this operator to the given operand.
```

- The familiar `Comparator` is also a **functional interface** with a single abstract method

```
int compare(T o1, T o2)
```

- How did we pass a `Comparator` object to, say `ArrayList.sort`?

IntStreams's map

- The usual way is to create a class that implements the `IntUnaryOperator` interface and override the `map` method

```
import java.util.function.IntUnaryOperator;

class MultiplyByTwo implements IntUnaryOperator {
    @Override
    public int applyAsInt(int x) {
        return 2 * x;
    }
}

        int sum = IntStream
            .rangeClosed(1, 10)
            .map(new MultiplyByTwo())
            .sum();
```


Anonymous Inner Class

- Rather than creating another class and pass an instance of the class to `map`, we can replace the argument with an anonymous inner class definition instead

```
        int sum = IntStream
            .rangeClosed(1, 10)
            .map(new IntUnaryOperator() {
                @Override
                public int applyAsInt(int x) {
                    return 2 * x;
                }
            })
            .sum();
```

- Which part of the anonymous inner class is *really* the useful bit? Can we simplify it?

Anonymous Method: Lambda Expression

- Class and method names (`IntUnaryOperator` and `applyAsInt`) do not add value
- Use an *anonymous method* without a name

```
        int sum = IntStream
            .rangeClosed(1, 10)
            .map((int x) -> { return 2 * x; })
            .sum();
```

- Lambda expression (Lambda): `(int x) -> {return 2 * x;}`
  - receives an integer parameter `x` and returns that value multiplied by two, much like

```
    int applyAsInt(int x) {
        return 2 * x;
    }
```


Lambda Expression

- Lambda syntax: *(parameterList) -> {statements}*
- Lambda does not require a method name, and the compiler infers the return type
- Other lambda variants:
  - `(x) -> {return 2 * x;}`: compiler infers parameter type
  - `(x) -> 2 * x`: body contains a single expression
  - `x -> 2 * x`: only one parameter
  - `() -> System.out.println("Lambdas!!!")`
- Methods can now be treated as data! 😎
  - pass lambdas as arguments to other methods (like `map`)
  - assign lambdas to variables for later use
  - return lambdas from methods

Intermediate and Terminal Operations

- Intermediate operations (like `map`) use **lazy evaluation**
- Does not perform any operations on stream's elements until a terminal operation is called, e.g. when filtering
  - Select elements that match a condition, or **predicate**

```
int sum = 0;
for (int x = 1; x <= 10; x++) {
    if (x % 2 == 0) {
        sum += (2 * x);
    }
}
```

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> 2 * x)
    .sum();
```

  - `filter` receives a method that takes one parameter and returns a **`boolean`** result; if it is true the element is included in the resulting stream
- Terminal operation use **eager evaluation**, i.e. perform the requested operation when they are called

Stream Elements

- Each intermediate operation results in a new stream
- Each new stream is an object representing the processing steps that have been specified up to that point in the pipeline
  - Chaining intermediate operations adds to the set of processing steps to perform on each stream element
  - The last stream object contains all processing steps to perform on each stream element
- When initiating a stream pipeline with a terminal operation, the intermediate operations' processing steps are applied one stream element after another
- Stream elements within a stream can only be consumed once
  - Cannot iterate through a stream multiple times

Stream Elements

- For following illustrates the movement of stream elements

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(
        x -> {
            System.out.println("filter: " + x);
            return x % 2 == 0;
        })
    .map(
        x -> {
            System.out.println("map: " + x);
            return 2 * x;
        })
    .sum();
System.out.println(sum);
```

```
filter: 1
filter: 2
map: 2
filter: 3
filter: 4
map: 4
filter: 5
filter: 6
map: 6
filter: 7
filter: 8
map: 8
filter: 9
filter: 10
map: 10
sum is 60
```


Method References

- A lambda that simply calls another method can be replaced with just that method's name, e.g. in the `forEach` terminal

```
IntStream
    .rangeClosed(1, 10)
    .forEach(x -> System.out.println(x));
```

- Using method reference

```
IntStream
    .rangeClosed(1, 10)
    .forEach(System.out::println);
```

- Types of method references:
  - reference to a static method
  - reference to an instance method
  - reference to a constructor

`IntStream` Operations for Arrays

- Consider the typical array operations below

```
int[] values = {7, 9, 5, 2, 8, 4, 1, 6, 10, 3};

int count = 0;
int min = values[0];
int max = values[0];
int sum = 0;
for (int x : values) {
    count++;
    if (x < min) {
        min = x;
    }
    if (x > max) {
        max = x;
    }
    sum += x;
}
double average = 1.0 * sum / values.length;
System.out.println("count: " + count);
System.out.println("sum: " + sum);
System.out.println("min: " + min);
System.out.println("max: " + max);
System.out.println("average: " + average);
```


`IntStream` Operations for Arrays

- Using `IntStream` operations

```
int[] values = {7, 9, 5, 2, 8, 4, 1, 6, 10, 3};

System.out.println("count: " +
        IntStream.of(values).count());
System.out.println("sum: " +
        IntStream.of(values).sum());
System.out.println("min: " +
        IntStream.of(values).min().getAsInt());
System.out.println("max: " +
        IntStream.of(values).max().getAsInt());
System.out.println("average: " +
        IntStream.of(values).average().getAsDouble());
```

- `IntStream.of(values)` creates an `IntStream` from the array `values`
- `min`, `max` returns `OptionalInt`; `average` returns `OptionalDouble`
- Use `getAsInt()` and `getAsDouble()` correspondingly since we know there are elements in the stream

User-defined Reductions

- Using `IntStream`'s `reduce` method
- Terminal operations are specific implementations of `reduce`
- For example, using `reduce` in place of `sum`

```
IntStream
    .of(values)
    .reduce(0, (x, y) -> x + y)
```

  - First argument to `reduce` is the operation's identity value
  - Second argument is the lambda that receives two `int` values, adds them and returns the result; in the above
    - First calculation uses identity value 0 as left operand
    - Subsequent calculations uses the result of the prior calculation as the left operand
    - If stream is empty, the identity value is returned

Boolean Terminal Operations

- Useful terminal operations that return a **`boolean`** result
  - `noneMatch` returns **`true`** if none of the elements pass the given predicate
  - `allMatch` returns **`true`** if every element passes the given predicate
  - `anyMatch` returns **`true`** if at least one element passes the given predicate
- Example: primality checking using external iteration

```
static boolean isPrime(int n) {
    for (x = 2; x < n; x++) {
        xf (n % x == 0) {
            return false;
        }
    }
    return true;
}
```

Boolean Terminal Operations

- Using streams

```
static boolean isPrime(int n) {
    return IntStream
        .range(2, n)
        .noneMatch(x -> n % x == 0);
}
```

- How about finding the first 500 prime numbers?

```
static void fiveHundredPrime() {
    int count = 0;
    int i = 2;
    while (count < 500) {
        if (isPrime(i)) {
            System.out.println(i);
            count++;
        }
        i++;
    }
}
```


Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
- Since streams are lazy until a terminal operation is performed, intermediate operations can be used to restrict the total number of elements in the stream
- `iterate` generates an ordered sequence starting using the first argument as a seed value

```
static void fiveHundredPrime() {
    IntStream
        .iterate(2, x -> x+1)
        .filter(x -> isPrime(x))
        .limit(500)
        .forEach(System.out::println);
}
```


Infinite Stream to Finite Stream

- Several intermediate operations convert an infinite stream to a finite stream
  - `limit` takes in an `int` `n` and returns a stream containing the first `n` elements of the stream
  - `takeWhile` takes in a predicate and returns a stream containing the elements of the stream, until the predicate becomes false; the resulting stream might still be infinite if the predicate never becomes false

```
static void primesLessThanFiveHundred() {
    IntStream
        .iterate(2, x -> x+1)
        .filter(x -> isPrime(x))
        .takeWhile(x -> x <= 500)
        .forEach(System.out::println);
}
```


Lecture Summary

- Appreciate the declarative style of programming using `IntStream`
- Understand how Java Functional Interface with a single abstract method can be used in stream operations
- Familiarity with writing lambda expressions as anonymous methods/functions
- Appreciate how lazy evaluations are used for intermediate operations, eager evaluation for terminal operations
- Know how to define reductions for use in a stream pipeline
- Appreciate how lazy evaluations support infinite streams