CS2030 Lecture 9

Java Streams

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

Semester 2 2022 / 2023

Lecture Outline and Learning Outcomes

- Know how to create **stream** pipelines for *internal* iteration
  - Know the difference between primitive and generic streams
  - Know how to write correct streams that are non-interfering
- Understand **lazy evaluation** in source/intermediate operations, and **eager evaluation** for terminal operations
- Appreciate how lazy evaluation supports **infinite stream**
- Able to implement a basic lazy context by encapsulating a `Supplier` functional interface for *delayed data*
  - Understand how `Lazy::map` can be implemented lazily
- Appreciate the concept of the lambda closure

External Iteration

- An external iteration is defined imperatively
  - e.g. sum of all integers in the closed interval $[1, 10]$

```
jshell> int sum = 0
sum ==> 0

jshell> for (int x = 1; x <= 10; x = x + 1) {
   ...>     sum = sum + x;
   ...> }

jshell> sum
sum ==> 55
```

- Errors could be introduced when
  - sum is initialized wrongly before the loop
  - looping variable x is initialized wrongly
  - loop condition is wrong
  - increment of x is wrong
  - aggregation of sum is wrong

Internal Iteration: `Stream`

- Internal iteration is defined declaratively
  - e.g. using a primitive integer stream

```
jshell> int sum = IntStream.rangeClosed(1, 10).
   ...> sum()
sum ==> 55
```

- Literal meaning "loop through values 1 to 10, and sum them"
- No need to specify how to iterate through elements or use any *mutable* variables — no variable state, no surprises! 😄
- A **stream** is a sequence of elements on which tasks are performed; stream elements move through a sequence of tasks in the stream pipeline
  - E.g. `sum` is assigned with the result of the stream pipeline

Stream Pipeline

- A stream pipeline comprises
  - a **data source**, e.g. `IntStream::rangeClosed` that starts the stream
  - some **intermediate operations**, e.g. `IntStream::map` that specify tasks to perform on a stream's elements

    ```
    jshell> IntStream stream = IntStream.rangeClosed(1, 10).map(x -> x * 2)
    stream ==> java.util.stream.IntPipeline$Head@12edcd21
    ```

  - a **terminal operation**, e.g. `IntStream::sum` that *reduces* the stream elements into a single value

    ```
    jshell> stream.sum()
    $.. ==> 110
    ```

- Each source/intermediate operation returns a new stream of processing steps specified up to that point in the pipeline
- Stream elements within a stream *can only be consumed once*

Reducing a Stream to a Value

- Iterate through `IntStream` elements and reduce to an **`int`**

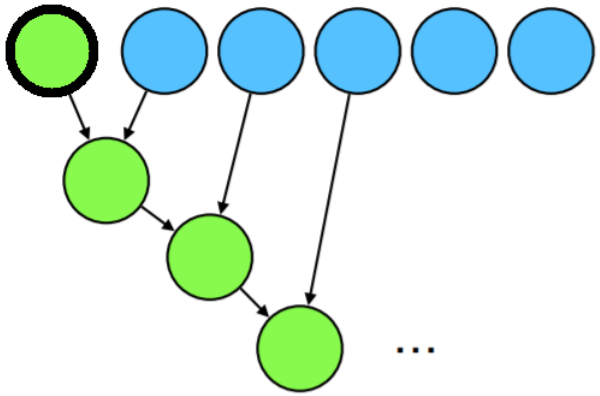
SAM

```
int reduce(int identity, IntBinaryOperator op)
```

- `IntBinaryOperator` with single abstract method:

```
int applyAsInt(int left, int right)
```

```
jshell> IntStream.rangeClosed(1, 10).
   ...> reduce(0, (x,y) -> x + y)
$.. ==> 55

jshell> IntStream.rangeClosed(1, 10).
   ...> reduce(1, (x,y) -> x * y)
$.. ==> 3628800
```



- Alternative one argument `reduce` that returns `OptionalInt`

OptionalInt as if stream is empty -> returns optional.empty

```
OptionalInt reduce(IntBinaryOperator op)
```

```
jshell> IntStream.range(1, 10).reduce((x, y) -> x < y ? x : y)
$.. ==> OptionalInt[9]

jshell> IntStream.range(1, 1).reduce((x, y) -> x < y ? x : y)
$.. ==> OptionalInt.empty
```

`flatMap` Method in Stream

- How about nested loops?

```
for (x = 1; x <= 3; x++)
   for (y = x; y <= 3; y++)
       System.out.println((x * y) + " "); // output is 1 2 3 4 6 9
```

- `map` tries to map each stream element into one other stream

```
jshell> IntStream.of(1,2,3).
   ...> map(x -> IntStream.rangeClosed(x,3).map(y -> x * y))
|  Error:
|  incompatible types: bad return type in lambda expression
|      java.util.stream.IntStream cannot be converted to int
|  map(x -> IntStream.rangeClosed(x,3).map(y -> x * y))
|           ^-----------------------------------^
```

- `flatMap` transforms each stream element into a stream of other elements (either zero or more) by taking in a function that produces another stream, and then *flattens* it

```
jshell> IntStream.of(1,2,3).
   ...> flatMap(x -> IntStream.rangeClosed(x,3).map(y -> x * y)).
   ...> forEach(x -> System.out.print(x + " "))
1 2 3 4 6 9
```

Generic `Stream<T>`

- `Stream<T>` is a stream over reference-typed objects, e.g.

```
jshell> int sum = Stream.<Integer>iterate(1, x -> x <= 10, x -> x + 1).
   ...> reduce(0, (x,y) -> x + y)
sum ==> 55
```

int to Integer

- `boxed()` wraps stream elements in its wrapper type

```
jshell> Stream<Integer> stream = IntStream.rangeClosed(1, 10).boxed()
stream ==> java.util.stream.IntPipeline$1@5010be6

jshell> List<Integer> list = stream.toList()
list ==> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- `mapToObj` converts from primitive to generic stream

```
jshell> IntStream.rangeClosed(1, 10).
   ...> mapToObj(x -> "<" + x + ">").
   ...> toList()
$.. ==> [<1>, <2>, <3>, <4>, <5>, <6>, <7>, <8>, <9>, <10>]
```

In this example,
converting from Integer to Object (String)

- `Stream::toList()` converts generic stream to generic list
- `List::stream()` converts generic list to generic stream

Correctness of Streams

- To ensure correct execution, stream operations
  - must not interfere with stream data

```
jshell> List<String> list = new ArrayList<String>(
   ...>    List.of("abc","def","xyz"))
list ==> [abc, def, xyz]

jshell> list.stream().peek(str -> {
   ...>    if (str.equals("xyz")) { list.add("pqr"); }
   ...> }).forEach(x -> {})
|  Exception java.util.ConcurrentModificationException
|  ...
```

  - *preferably* stateless with no side effects
    - operations like `filter` and `map` are **stateless**, i.e. processing one stream element does not depend on other stream elements
    - **stateful** operations like `sorted`, `limit`, `distinct`, etc. depend on the current state

Lazy Evaluation in Streams

- Source/intermediate operations use **lazy evaluation**
  - does not perform any operations on stream's elements until a terminal operation is called

numbers are only generated when it is needed (terminal operations called)

- Terminal operations use **eager evaluation**
  - performs the requested operation as soon as it is called

```
jshell> Stream.<Integer>iterate(1, x -> x + 1).
   ...> limit(5).
   ...> peek(x -> System.out.println("limit: " + x)).
   ...> filter(x -> x % 2 == 0).
   ...> peek(x -> System.out.println("filter: " + x)).
   ...> map(x -> 2 * x).
   ...> peek(x -> System.out.println("map: " + x)).
   ...> reduce(0, (x,y) -> x + y)
```

```
limit: 1
limit: 2
filter: 2
map: 4
limit: 3
limit: 4
filter: 4
map: 8
limit: 5
$.. ==> 12
```

Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
  - `Stream<T>::generate(Supplier<T> supplier)` produces an infinite sequence of values generated by **`supplier`**
  - `Stream<T>::iterate(T seed, UnaryOperator<T> next)` produces an infinite sequence by repeatedly applying the function **`next`** starting with the **`seed`** value
- Intermediate operations, e.g. **`limit`**, can be used to restrict the total number of elements in the stream

```
jshell> Stream.<Integer>iterate(1, x -> x + 1).
   ...> filter(x -> x % 2 == 1).
   ...> limit(20). // find first 20 odd numbers
   ...> forEach(x -> System.out.print(x + " "))
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
```

Lazy Class

- To understand how lazy evaluation works, define a `Lazy` class

```
import java.util.function.Supplier;

class Lazy<T> implements Supplier<T> {
    private final Supplier<T> supplier;

    private Lazy(Supplier<T> supplier) {
        this.supplier = supplier;
    }

    static <T> Lazy<T> of(Supplier<T> supplier) {
        return new Lazy<T>(supplier);
    }

    static <T> Lazy<T> of(T t) {
        return new Lazy<T>(() -> t);
    }

    @Override
    public T get() {
        return supplier.get();
    }
}
```

```
jshell> int foo() {
   ...>     System.out.println("foo");
   ...>     return -1;
   ...> }
|  created method foo()

jshell> Lazy<Integer> lazy = Lazy.of(foo())
foo
$.. ==> Lazy@ae45eb6

jshell> lazy.get()
$.. ==> -1

jshell> lazy = Lazy.<Integer>of(() -> foo())
$.. ==> Lazy@6f7fd0e6

jshell> lazy.get()
foo
$.. ==> -1
```

foo() is evaluated before being passed in Lazy.of(-1)

- `Lazy.of(foo())` evaluates `foo` method *eagerly*
- `Lazy.of(() -> foo())` evaluates `foo` *lazily*, i.e. only when `get()` is invoked sometime later

Mapping a Lazy Value

- Define `map` that returns a new `Lazy`

```
<R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
    Supplier<R> supplier = () -> mapper.apply(this.get());
    return Lazy.<R>of(supplier);
}
```

mapper.apply is inside the supplier : delayed

```
// Lazy.of(() -> foo()).map(x -> x+1)
jshell> Lazy<Integer> i = Lazy.<String>of(() -> "abc").
   ...>    map(x -> { System.out.println("map1"); return x.length(); }).
   ...>    map(x -> { System.out.println("map2"); return x * 2; })
i ==> Lazy@51565ec2 // map is not evaluated until a get()

jshell> i.get() // map is lazily evaluated :)
map1
map2
$.. ==> 6
```

won't work

- What about the following implementation of `map`?

```
<R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
    R r = mapper.apply(this.get());
    return Lazy.<R>of(() -> r);
}
```

when you construct the map, you are already doing a mapper.apply()

Lambda Closure

- Lambdas declared inside a method are also *local classes*

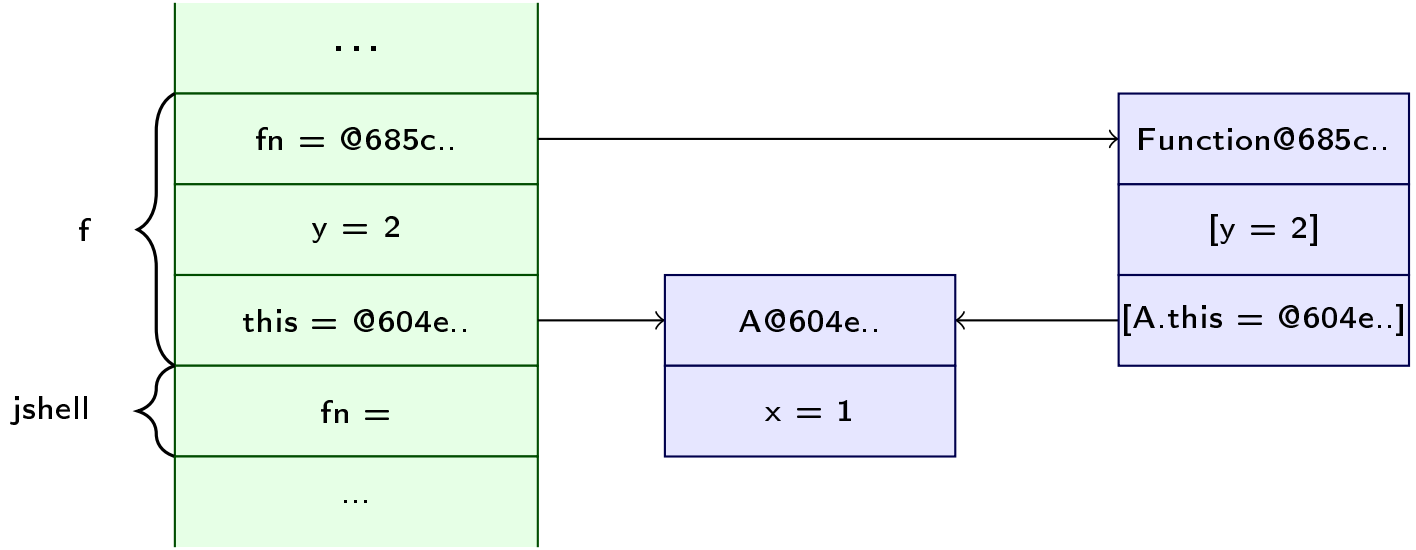
```
jshell> class A {
   ...>    private final int x;
   ...>    A(int x) {
   ...>       this.x = x;
   ...>    }
   ...>    Function<Integer,Integer> f(int y) {
   ...>       return z -> this.x + y + z; // or A.this.x + y + z ?
   ...>    }
   ...> }
|  created class A

jshell> Function<Integer,Integer> fn = new A(1).f(2)
fn ==> A$$Lambda$14/1196765369@26be92ad

jshell> fn.apply(3)
$.. ==> 6
```

- *Lambda closure*: lambda expression closes over it's enclosing method and class
  - captures the variables of the enclosing method and reference to the enclosing class

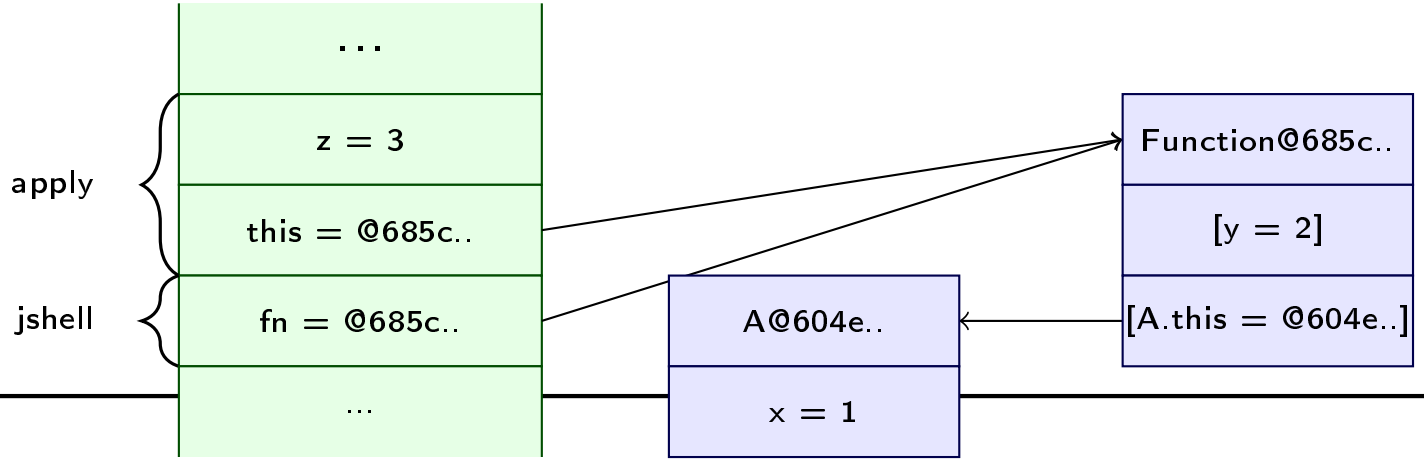
Java Memory Model

- Memory model just before returning from the method `f`

```
jshell> Function<Integer,Integer> fn = new A(1).f(2)
```



- Memory model upon invoking the method `fn.apply(3)`



Exercise

- Let's repeat the exercise in the previous lecture but with method `f` returning a lambda expression instead

```
jshell> class A {
   ...>     Integer apply(int x) {
   ...>         return x * 10;
   ...>     }
   ...>     Function<Integer,Integer> f(int y) {
   ...>         return z -> A.this.apply(z) + y;
   ...>     }
   ...> }
|  modified class A
```

(3*10) + 2 = 32

- What is the outcome of `new A().f(2).apply(3)`?
- Now replace `A.this.apply(z)` in method `foo` with `this.apply(z)`. Does it compile?

infinite loop as it is constantly invoking own apply method

  - what is the outcome of `new A().f(2).apply(3)` now?
  - what is the difference as compared to returning an anonymous inner class?