CS2030 Lecture 9

The Art of Being Lazy

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

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- Understand the differences between **lazy evaluation** and eager evaluation
- Able to implement basic lazy evaluation with **caching**
- Able to use `Supplier` functional interface for *delayed data*
- Awareness of *variable capture* associated with a *local class*
- Appreciate how a lazy list (or infinite list) can be implemented with an awareness of *upstream/downstream* movement supported by method invocation and return

Eager Evaluation

- Consider the following:

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

jshell> int bar() {
   ...> System.out.println("bar");
   ...> return 2;
   ...> }
|  created method bar()

jshell> int baz(int n, int x, int y) {
   ...> return n % 2 == 0 ? x : y;
   ...> }
|  created method baz(int,int,int)
```

```
jshell> baz(1,foo(),bar())
foo
bar
$.. ==> 2

jshell> baz(2,foo(),bar())
foo
bar
$.. ==> 1
```

- Notice that both `foo()` and `bar()` have to be evaluated before passing the values to `baz()`
  - but `baz()` need only return one of the evaluations

Lazy Evaluation

- Alternatively, pass the functionalities of `foo()` and `bar()` to the `baz()` method and let it decide which of the two to invoke
- Use `Supplier<T>` with SAM `T get()` as *delayed data*

```
jshell> Supplier<Integer> supplier = () -> foo()
supplier ==> $Lambda$14/575593575@14acaea5

jshell> supplier.get()
foo
$.. ==> 1

jshell> int baz(int n, Supplier<Integer> x, Supplier<Integer> y) {
   ...> return n % 2 == 0 ? x.get() : y.get();
   ...> }
|  created method baz(int,Supplier<Integer>,Supplier<Integer>)

jshell> baz(1, () -> foo(), () -> bar())
bar // only bar method is invoked
$.. ==> 2

jshell> baz(2, () -> foo(), () -> bar())
foo // only foo method is invoked
$.. ==> 1
```

Caching

- Now consider this:

```
jshell> baz(1, () -> foo(), () -> bar())
bar
$.. ==> 2

jshell> baz(2, () -> foo(), () -> bar())
foo
$.. ==> 1
```

- Assuming no *side effects*, i.e.
  - `foo()`/`bar()` is not affected by changes in external states
  - `foo()`/`bar()` will always return the same value
- Avoid invoking `foo()`/`bar()` repeatedly to get the same value
  - need a context to handle a *cached supplier* — `Lazy`
    - `Supplier` to handle delayed data
    - `Optional` to store the cache value

Lazy Class *without* **Caching**

- Start by defining a `Lazy` class without caching

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

class Lazy<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 value) {
        return new Lazy<T>(() -> value);
    }

    T get() {
        System.out.println("In method get..."); // added for instrumentation purposes
        return supplier.get();
    }
}
```

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

jshell> Lazy.<Integer>of(foo()).get()
foo
In method get...
$.. ==> 1

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

jshell> Lazy.<Integer>of(() -> foo()).get()
In method get...
foo
$.. ==> 1
```

- `Lazy.of(foo())` evaluates `foo` method before passing
- On the other hand, `Lazy.of(() -> foo())` evaluates `foo` only when the `Lazy::get()` method is invoked

Lazy Class *with* Caching

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

class Lazy<T> {
    private final Supplier<T> supplier;
    private Optional<T> cache; // not declared final

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

    static <T> Lazy<T> of(Supplier<T> supplier) {
        // throws an exception if supplier is null
        return new Lazy<T>(Optional.<Supplier<T>>ofNullable(supplier).orElseThrow())
    }

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

    T get() {
        T v = this.cache.orElseGet(this.supplier);
        this.cache = Optional.<T>of(v); // mutating the property?
        return v;
    }
}
```

Lazy Class *with* Caching

- `Lazy::get()` first tests if a value is present in `cache`
  - if it is, assign local variable `v` with the value;
  - otherwise, invoke `supplier`'s `get` method through `Optional::orElseGet` and assign the value to `v`
- `v` is then wrapped in an `Optional` and assigned to `cache`
  - this relies on the `cache` being mutable
  - however, to the client, `Lazy` is still *observably immutable*

```
jshell> Lazy<Integer> lazy = Lazy.of(() -> foo());
lazy ==> Lazy@91161c7

jshell> lazy.get() // lazy is observably immutable if
foo                // foo has no side-effects (e.g. output)
$.. ==> 1

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

Mapping a Lazy Value

- Define `Lazy::map` as follows makes it eagerly evaluated

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

```
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; })
map1
map2
i ==> Lazy@51565ec2 // map is eagerly evaluated
```

- Redefine `map` to create a new `Lazy` wrapped in a `Supplier`

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

```
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()
map1
map2
$.. ==> 6
```

Local Class and Variable Capture

- Lambdas and anonymous classes declared inside a method are called *local classes*
- Cannot mutate client-side states inside local classes

```
jshell> boolean isPrime(int n) {
   ...>    boolean prime = true;
   ...>    IntStream.range(2, n)
   ...>       .forEach(x -> { if (n % x == 0) prime = false; });
   ...>    return n > 1 && prime;
   ...> }
|  Error:
|  local variables referenced from a lambda expression must be final
|  or effectively final
|  .forEach(x -> { if (n % x == 0) prime = false; });
```

- Java only allows a local class to access variables that are explicitly declared final or effectively (or implicitly) final
  - An implicitly final variable is one that does not change after initialization

Local Class and Variable Capture

- Rewriting `Supplier` in `Lazy::get` as anonymous inner class

```
<R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
    Supplier<R> supplier = new Supplier<R>() {
        @Override
        public R get() {
            return mapper.apply(Lazy.this.get()); // this.get()? or get()?
        }       // mapper and Lazy is accessible from Supplier local class
    };
    return Lazy.<R>of(supplier);
}
```

- *Closure*: local class closes over it's enclosing method and class
  - *variable capture*: local class makes a copy of variables of the enclosing method and reference to the enclosing class

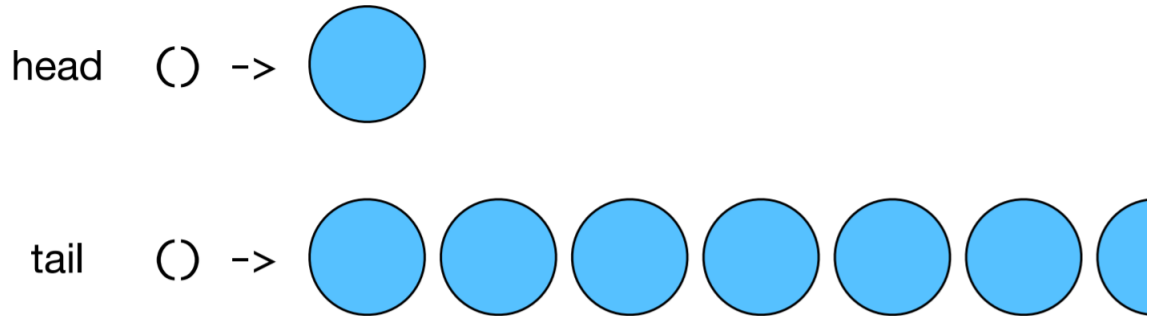
```
jshell> class A {
   ...> int x;
   ...> A(int x) { this.x = x; }
   ...> Function<Integer,Integer> f(int y) { return z -> A.this.x + y + z; } }
|  created class A

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

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

From `Lazy` to `LazyList`

- What defines a lazy (or infinite) list?
  - the head which is a value of type `T`
  - the tail which is a lazy list comprising elements of type `T`



- Since evaluation of `head` and `tail` is delayed until a terminal operation is called, need to wrap each in a `Supplier`

```
class LazyList<T> {
    private final Supplier<T> head;
    private final Supplier<LazyList<T>> tail;

    private LazyList(Supplier<T> head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

Data Source Operations

- Factory method `LazyList::generate`

```
static <T> LazyList<T> generate(Supplier<T> supplier) {
    Supplier<T> newHead = supplier;
    Supplier<LazyList<T>> newTail = () -> LazyList.<T>generate(supplier);
    return new LazyList<T>(newHead, newTail);
}
```

- Factory method `LazyList::iterate`

```
static <T> LazyList<T> iterate(T seed, Function<T,T> next) {
    Supplier<T> newHead = () -> seed;
    Supplier<LazyList<T>> newTail =
        () -> LazyList.<T>iterate(next.apply(seed), next);
    return new LazyList<T>(newHead, newTail);
}
```

- Every non-terminal operation returns a new `LazyList`
  - `newHead` specifies how to generate the element
  - `newTail` specifies how to generate the operation with respect to the next pipeline

Terminal Operation: `forEach`

- Generate elements with `head`/`tail` and (non-**`private`**) `get`

```
jshell> LazyList.<Integer>iterate(1, x -> x + 1)
$.. ==> LazyList@59f95c5d

jshell> LazyList.<Integer>iterate(1, x -> x + 1).head.get()
$.. ==> 1

jshell> LazyList.<Integer>iterate(1, x -> x + 1).tail.get()
$.. ==> LazyList@2c13da15

jshell> LazyList.<Integer>iterate(1, x -> x + 1).tail.get().head.get()
$.. ==> 2
```

- Defining the `forEach` terminal

```
    void forEach(Consumer<? super T> consumer) {
        LazyList<T> curr = this;
        for (int i = 0; i < 5; i++) {
            T value = curr.head.get();
            consumer.accept(value);
            curr = curr.tail.get();
        }
    }
```

```
jshell> LazyList.iterate(1, x -> x + 1).forEach(x -> System.out.println(x + " "))
1 2 3 4 5
```

Intermediate Operation: `map`

- `map`: each element is mapped to another element

```
<U> LazyList<U> map(Function<? super T, ? extends U> mapper) {
    Supplier<U> newHead = () -> mapper.apply(LazyList.this.head.get());
    Supplier<LazyList<U>> newTail = () -> LazyList.this.tail.get().map(mapper);
    return new LazyList<U>(newHead, newTail);
}
```

```
Iterate:           [() -> seed,       () -> LazyList.iterate(next.apply(seed), next)]
                        ^                  ^
                        |                  |
Map: [() -> mapper.apply(head.get()), () -> tail.get().map(mapper)]
```

- When `curr.head.get()` in the `forEach` terminal is invoked
  - initiates an upstream movement back via `map` to `iterate` to obtain the first element
  - the element then moves downstream (and processed) along all operations before reaching the terminal
- `curr.tail.get()` behaves similarly to move upstream, and constructs the new pipeline while it progresses downstream 😄

Other Intermediate Operations

- `limit`: *to start you off...*

```
LazyList<T> limit(int n) {
    if (n > 0) {
        return new LazyList<T>(() -> LazyList.this.head.get(),
            () -> LazyList.this.tail.get().limit(n-1));
    } else {
        ...
    }
}
```

- `filter`: missing value (`Optional`) when element is filtered-off
  - whether an element is filtered or otherwise, requires probing the next generated value twice, when
    - the terminal initiates upstream `get`s for next element
    - the new pipeline is constructed
  - need to cache the value (`Lazy`) when it is first generated