Lecture 8: Lambdas and Streams

Learning Objectives

After this lecture, students should be familiar with:

- the concept of closure and its relation to lambda expressions
- the concept of eager evaluation vs. lazy evaluation
- Java `Optional` class and its operations
- the concept of function as delayed data and its application in defining an infinite list
- Java `Stream` class and its operations
- using the stream operations to write declarative-style code, avoiding loops and branches

We continue where we left off in Lecture 7.

Lambda as Closure

Just like a local class and an anonymous class, a lambda expression can capture the variables of the enclosing scope. Recall that, a lambda expression is just a shorthand to an anonymous class after all.

For instance, if you do not wish to generate the service time of a customer at the time of arrival, you can pass in a `Supplier` to `Customer` instead:

```
Customer c = new Customer(() -> rng.GenServiceTime());
```

Here, `rng` is a variable captured from the enclosing scope.

Just like in local and anonymous classes, a captured variable must be either explicitly declared as `final` or is effectively final.

A lambda expression, therefore, stores more than just the function to invoke -- it also stores the data from the environment where it is defined. We call such construct that stores a function together with the enclosing environment a *closure*.

Function as Cross-Barrier State Manipulator

We have seen that functional-style programming allow us to do a few things that we couldn't before with functions: (i) we can assign function to a variable, pass functions around, return it from another function; (ii) we can compose and create functions dynamically during runtime; (iii) we can partially evaluate a function.

Let's take a look at two ways functional-style programming helps us write better programs.

We have seen the `applyList` method last week, where we pass a `Function<T,R>` object to manipulate the items in the list. This method, commonly known as `map`, saves us from writing loops and leads to shorter and less buggy code. If we view the internal representation of the list of items as behind the abstraction barrier, then we are manipulating data behind the abstraction barrier without knowing the internals of the object -- something we could only do through the interfaces provided by the implementer earlier, before the introduction of functions.

`Optional`

Another way passing in functions to manipulate internal data is helpful is the `Optional<T>` [https://docs.oracle.com/javase/8/ docs/api/java/util/Optional.html] class. `Optional<T>` is a wrapper around a reference to either a `T` object or a `null`. Recall that we said bugs can occur if we write functions that return a value not in its codomain, and we gave `null` as an example. Often, we write functions that return `null` to indicate a special situation (e.g., `server = shop.findIdleServer()` cannot find an idle server) but use the returned reference without checking for `null` (e.g., `server.serve(customer)`) leading to `NullPointerException` being raised, because `null` is not a Server object.

Wrapping the returned reference with an `Optional` changes the codomain of the function written, as `null` is now explicitly in the codomain.

```
Optional<Server> server = shop.findIdleServer();
```

We now have a reference to an `Optional` object, which wraps around either a server or `null`. We cannot call `server.serve(customer)` since the actual reference to the server is behind the abstraction barrier. The `Optional` class provides a method `ifPresent` that takes in a `Consumer`. So you can call:

```
server.ifPresent(s -> s.serve(customer))
```

If server wraps around `null`, then `ifPresent` do nothing. Using `Optional` means that we no longer have to write branching statements:

```
   server = shop.findIdleServer();
   if (server != null) {
     server.serve(customer);
   }
```

It can makes code with multiple-level of branching clearer. Without `Optional`,

```
    server = shop.findIdleServer();
    if (server == null) {
      server = shop.findShortestQueue();
      if (server == null) {
         customer.leave();
      } else {
         server.serve(customer);
      }
    } else {
      server.serve(customer);
    }
```

Using `Optional`, we write

```
shop.findIdleServer()
    .or(shop::findShortestQueue)
```

```
    .ifPresentOrElse(
        s -> s.serve(customer),
        customer::leave);
```

Java 8 vs. 9
`or` and `ifPresentOrElse` are available in Java 9 only.

The branching logic still exists, but is internalized, just like we have internalized loops with `applyList` method. Furthermore, we do not explicitly compares with `null` anymore and this code will never raise a `NullPointerException`.

Updated Notes
This whole section is more or less rewritten since published. If you print your notes, please make sure that you have the latest version.

Initializing an `Optional`

To wrap up the discussion, let's see how we can create an `Optional` object. If you want to wrap a non-`null` value in an `Optional`, call `Optional.of(value)`. Otherwise, if you want to wrap it in a `null`, call `Optional.empty()`.

Alternatively, if you do not want to check if the value is `null` or not, call `Optional.ofNullable(value)` which will return one of the above appropriately for you.

`Optional` **In other languages**
Scala has `Option`; Haskell has `Maybe`. If you use Python, check out the `PyMonad` library that supplies a `Maybe` class.

Function as Delayed Data

Consider a function that produces a new value or values. We can consider the function as a promise to provide us the given data sometime later, when needed. For instance:

```
() -> rng.genServiceTime()
```

is not the value of a service time, but rather, a supplier of the service time. We invoke this supplier only when we need the service time.

Consider the case where the function to generate data is an expensive one. We can delay the execution of the expensive function until it is absolutely needed. This is called *lazy evaluation*.

An Infinite List

Lazy evaluation allows us to build data structures that we could not before. For instance, we can create and manipulate a list that is infinitely long.

How can we represent and manipulate an infinitely long list? If we store the values of each element in the list, then we will run out of memory pretty soon. If we try to manipulate every element in the list, then we will enter an infinite loop.

The trick to building an infinite list, is to treat the elements in the list as *delayed data*, and store *a function that generates the elements*, instead of the elements themselves.

We can think of an infinite list as consisting of two functions, the first is a function that generates the first element, and the second is a function that generates the rest of the list.

```
class InfiniteList<T> {
  private Supplier<T> headSupplier;
  private Supplier<InfiniteList<T>> tailSupplier;

  public InfiniteList(Supplier<T> headSupplier, Supplier<InfiniteList<T>> tailSupplier) {
    this.headSupplier = headSupplier;
    this.tailSupplier = tailSupplier;
  }

    :
}
```

We can then construct an infinite list in different ways by passing in different suppliers.

Suppose we want every element in the list to be generated using the same supplier. We can write a method that does as follows:

```
  public static <T> InfiniteList<T> generate(Supplier<T> supply) {
    return new InfiniteList<T>(supply,
        () -> InfiniteList.generate(supply));
  }
```

Or we can construct an infinite list consisting of a sequence of elements, each computed from the previous element using a `next` function:

```
  public static <T> InfiniteList<T> iterate(T init, Function<T, T> next) {
    return new InfiniteList<T>(() -> init,
      () -> InfiniteList.iterate(next.apply(init), next));
  }
```

Here are some examples of how to use the two methods above:

```
InfiniteList<Integer> ones = InfiniteList.generate(() -> 1); // 1, 1, 1, 1, ....
InfiniteList<Integer> even = InfiniteList.iterate(0, x -> x + 2); // 0, 2, 4, 6, ...
```

A list that is defined this way is lazily evaluated. We will not call the supplier to generate the elements until we need it -- this is in contrast to the eagerly evaluate `LambdaList` from the exercise in Lecture 7.

Let's see how we can manipulate this list. Consider the `findFirst` method, which returns the first element in the list that satisfies the given predicate.

```
public T findFirst(Predicate<T> predicate) {
  InfiniteList<T> list = this;
  while (true) {
    T next = list.headSupplier.get();
    if (predicate.test(next)) {
      return next;
    }
    list = list.tailSupplier.get();
  }
}
```

In the method above, we repeatedly invoke the supplier, until we find an element that matches the predicate. This way, we never had to generate every element in the list just to find the first element that matches.

Stream

Such a list, possibly infinite, that is lazily evaluated on demand is also known as a *stream*. Java 8 provides a class `Stream` and a set of useful and powerful methods on streams, allowing programmers to manipulate data very easily. Java 9 adds a couple of useful methods, `takeWhile` and `dropWhile`, which is also invaluable. To take full advantage of streams, we will be using Java 9, not Java 8 for the rest of this class.

Stream Operations

A few things to note before I show you how to use streams. First, the operations on streams can be classified as either *intermediate* or *terminal*. An *intermediate* operation returns another stream. For instance, `map`, `filter`, `peek` are examples of intermediate operations. An intermediate operation does not cause the stream to be evaluated. A terminal operation, on the other hand, forces the streams to be evaluated. It does not return a stream. `reduce`, `findFirst`, `forEach` are examples of terminal operation. A typical way of writing code that operates on streams is to chain a series of intermediate operation together, ending with a terminal operation.

Second, a stream can only be consumed once. We cannot iterate through a stream multiple times. We have to create the stream again if we want to do that:

```
Stream<Integer> s = Stream.of(1,2,3);
s.count();
s.count(); // <- error
```

In the example above, we use the `of` static method with a variable number of arguments to create a stream. We can also create a stream by:

- converting an array to stream using `Arrays.stream` method
- converting a collection to stream using `stream` method
- reading from a file using `Files.lines` method
- using the `generate` method (provide a `Supplier`) or `iterate` method (providing the initial value and incremental operation).

You have seen many of the stream operations before, in Question 5 of Exercise 7, including `map`, `reduce`, `filter`, and `forEach`. Even though they are in the context of an eagerly evaluated list, the semantics are the same. Here are a few more useful ones.

- `flatMap` is just like `map`, but it takes in a function that produces another stream (instead of another element), and it `flattens` the stream by inserting the elements from the stream produced into the stream.

Let see an example. The lambda below takes a string and return a stream of `Integer` objects:

```
x -> x.chars().boxed()
```

We can create a stream of strings using the static `of` method from `Stream`:

```
Stream.of("live", "long", "and", "prosper")
```

If we chain the two together, using `map`, however, we will produce a stream of stream of `Integer`.

```
Steam.of("live", "long", "and", "prosper")
    .map(x -> x.chars().boxed())
```

To produce a stream of `Integer`s, we use `flatMap()`:

```
Stream.of("live", "long", "and", "prosper")
    .flatMap(x -> x.chars().boxed())
```

- `sorted` is an intermediate operation that returns a stream with the elements in the stream sorted. Without argument, it sorts according to the natural order. You can also pass in a `Comparator` to tell `sorted` how to sort.
- `distinct` is another intermediate operation that returns a stream with only distinct elements in the stream.

`distinct` and `sorted` are stateful operations -- it needs to keep track of states in order to perform the operation. `sorted`, in particular, needs to know every element in the stream before it can output the result. They are also known as `bounded` operations, since they should only be called on a finite stream -- calling them on an infinite stream is a very bad idea.

Let's look at an example. The code below shows how we can print out the unique characters of a given sequence of streams in sorted order

```
Stream.of("live", "long", "and", "prosper")
    .flatMap(x -> x.chars().boxed())
    .distinct()
    .sorted()
    .map(Character::toChars)
    .forEach(System.out::print);
```

There are several intermediate operations that convert from infinite stream to 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.

Here are more useful terminal operations:

- `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.

To illustrate the use of the `Stream` class and its methods, let's look at an example.

Example: Is this a prime?

Consider the method below, which checks if a given `int` is a prime:

```
boolean isPrime(int x) {
  for (i = 2; i <= x-1; i++) {
```

```
    if (x % i == 0) {
      return false;
    }
  }
  return true;
}
```

The code coudln't be simpler -- or can it? With streams, we can write it as:

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

`IntStream` is a special `Stream` for primitive type `int`, the `range(x,y)` method generates a stream of `int` from `x` to `y-1`.

What if we want to print out the first 500 prime numbers, starting from 2? Normally, we would do the following:

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

The code is still considered simple, and understandable for many, but I am sure some of us will encounter a bug the first time we write this (either forgot to increment the counter, or put the increment in the wrong place). If you look at the code, there are a couple of components:

- Lines 3 and 9 deal with iterating through different numbers for primality testing
- Line 4 is the test
- Lines 2, 4, and 7, deal with limiting the output to 500 primes
- Line 5 is the action to perform on the prime

With streams, we can write it like the following:

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

Notice how each of the four components matches neatly with one operation on stream!

With stream, we no longer have to write loops, we have moved the iterations to within each operation in stream. We no longer need to maintain states and counters, they are done within each operation as needed as well. This has another powerful implication: our code become more *declarative*, we only need to concern about what we want at each step, much less about how to do it.

You should take a look at the methods provided by the `Stream` class, and read through the APIs, a few times, they formed the fundamental building blocks for writing functional-style data processing code in Java.

Exercise

1. Write your own `Optional` class with the following skeleton:

```
class Optional<T> {
  T value;

  public static <T> Optional<T> of(T v) {
    :
  }

  public static <T> Optional<T> ofNullable(T v) {
    :
  }

  public static <T> Optional<T> empty(T v) {
    :
  }

  public void ifPresent(Consumer<? super T> consumer) {
    :
  }

  public Optional<T> filter(Predicate<? super T> predicate) {
    :
  }

  public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    :
  }

  public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    :
  }

  public T orElseGet(Supplier<? extends T> other) {
    :
  }
}
```

2. Solve each of the following with Java 9 `Stream`.
   - Write a method `factors` with signature `LongStream factors(long x)` that takes in `long x` and return a `LongStream` consisting of the factors of `x`. For instance, factors(6) should return the stream 1, 2, 3, 6.
   - Write a method `primeFactors` with signature `LongStream primeFactors(long x)` that takes in `long x` and return a `LongStream` consisting of the prime factors of `x` (a prime factor is a factor that is a prime number, excluding 1). For instance, prime factors of 6 are 2 and 3.

- Write a method `omega` with signature `LongStream omega(int n)` that takes in an `int n` and return a `LongStream` containing the first n omega numbers [https://oeis.org/A001221]. The i-th omega number is the number of distinct prime factors for the number i. The first 10 omega numbers are 0, 1, 1, 1, 1, 2, 1, 1, 1, 2.

3. Write a method `product` that takes in two `List` objects `list1` and `list2`, and produce a `Stream` containing elements combining each element from `list1` with every element from `list2` using a given `BiFunction`. This operation is similar to a Cartesian product.

   For instance,

```
ArrayList<Integer> list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
Collections.addAll(list1, 1, 2, 3, 4);
Collections.addAll(list2, 10, 20);
product(list1, list2, (str1, str2) -> str1 + str2)
    .forEach(System.out::println);
```

   gives the output:

```
11
21
12
22
13
23
14
24
```

   The signature for `product` is

```
  public static <T,U,R> Stream<R> product(List<T> list1, List<U> list2,
      BiFunction<? super T, ? super U, R> f)
```

4. Write a method that returns the first n Fibonacci numbers as a `Stream<BigInteger>`. For instance, the first 10 Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. It would be useful to write a new `Pair<T, U>` class that keeps two items around in the stream. We use the `BigInteger` class to avoid overflow.