Lecture 9: Functional-Style Programming Patterns

Learning Objectives

After this lecture, students should:

- Understand what is a SAM and how to write own's functional interface that can be represented by a lambda expression
- Understand what are functors and monads in the context of Java's `Stream`, `Optional`
- Understand the laws that a functor and monad must obey and be able to verify them
- Aware of different programming patterns using lambda, including functor, monad, polymorphism, and observer.

Functional Interface

We have seen how we can write lambda expressions of types `Function`, `Predicate`, `Supplier`, `Consumer`, and `BiFunction`. We, however, are not limited the types defined in `java.util.function`. We can use a lambda expression as a short hand to an anonymous class that implements any interface with *a single abstract method*. The reason there has to be only one abstract method is so that the compiler can infer which method body the lambda expression implements. Such an interface is more commonly known as a *SAM interface*.

A SAM interface can contain multiple methods, but only one needs to be abstract, others can contain `default` implementations.

For instance, Java has the following interface:

```
interface Runnable {
  void run();
}
```

There is only one method, and it is abstract (no default implementation). So it is a valid SAM interface.

We can write:

```
Runnable r = () -> { System.out.println("hello world"); }
```

We can annotate a class with `@FunctionalInterface` to hint our intention to the compiler and to let the compiler helps us catch any unintended error, such as when we add a second abstract method to the interface.

We can define our own functional interfaces as well. For instance, we can define:

```
@FunctionalInterface
interface FindServerStrategy {
  Server findQueue(Shop shop);
}

FindServerStrategy greedy = shop -> shop.getShortestQueue();
FindServerStrategy typical = shop -> shop.getFirstAvailableQueue();
```

While the interface above can be represented with a `Function<Shop,Server>`, one might find it easier to read

```
greedy.findQueue(shop)
```

as opposed to:

```
greedy.apply(shop)
```

Functor

In this lecture, we are going to abstract out some useful patterns that we have seen so far in functional-style programming in Java, and relates it to concepts in functional programming.
Once you see and understand the patterns, hopefully you can reapply the patterns in other context.

Let's start with a simple one, called *functor*. This funny name originated from a branch of mathematics, called category theory. We can think of a functor as something that takes in a function and returns another functor. We can think of it the interface below:

```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

A functor can be any class that implements the interface above, or matches the pattern above.

Let's took at the example below:

```
class A {
  private int x;

  public A(int i) {
    this.x = i;
  }

  public A f(Function<Integer,Integer> func) {
    if (this.x > 0) {
      return new A(func.apply(x));
    } else {
      return new A(0);
    }
  }

  public boolean isSameAs(A a) {
    return this.x == a.x;
  }
}
```

The class `A` above takes in a function and returns another `A` with `func` applied on the content `x`, if `x` is positive. Otherwise, it returns another `A` with 0.

Despite that it does not implement the interface `Functor`[1][#fn:1], it does match the pattern of having a method that takes in a function and returns itself, it is a special case since both `R` and `T` are `Integer`.

Functor Laws

Matching the patterns syntactically, however, is not enough to be a functor. A functor has to semantically obey the functor laws, which are:

- if `func` is an identity function `x -> x`, then it should not change the functor.
- if `func` is a composition of two functions $g \cdot h$, then the resulting functor should be the same as calling `f` with h and then with g.

Let's check:

```
A a = new A(-1);
a.isSameAs(a.f(x -> x));
a.f(x -> x + 1).f(x -> x * 2).isSameAs(a.f(x -> (x + 1) * 2));
```

The second line above failed to return true. As such, class `A` violates the first functor law and therefore is not a functor. Class `B` below, however, is a functor:

```
class B {
  private int x;

  public B(int i) {
    this.x = i;
  }

  public B f(Function<Integer,Integer> func) {
    return new B(func.apply(x));
  }

  public boolean isSameAs(B a) {
    return this.x == a.x;
  }
}
```

It is easy to see that if `func` is `x -> x`, then `B(func.apply(x))` is just `B(x)`. Further, if `func` is `g.compose(h)`, then calling `func.apply(x)` is the same as `g.apply(h.apply(x))`.

Another way to think of a functor, in the OO-way, is that that it is *a variable wrapped within a class in some context*. Instead of manipulating the variable directly, we pass in a function to the class to manipulate the variable. The variable must then interact with the function as if it is not be wrapped, and the class should not interfere with the function (as in the class `A`). In other words, we use lambda expression for *cross-abstraction barrier manipulation*.

You have actually seen several functors before. You might recognize by now that `f` is just our old friend `map`. `LambdaList` (from Lecture 7), `InfiniteList` (from Lab 4), and `Stream` (from `java.util.stream`) are functors wrap around a (possibly infinite) list of items.

Functors in other languages
Haskell, Scala, Python, Javascript, and other functional languages have functors as well. C++, unfortunately, uses the term functors to mean function object -- a function object is not a functor in the sense of the word in category theory. So, do not get confused between the two.

Once you understand the laws of functor and recognize this pattern, it is easy to learn about new classes -- one just have to tell you that it is a functor, and you will know how the class should behave. For instance, I can tell you that `Optional` is a functor. Do you know what method `Optional` supports and how it behave?

Recall that you can wrapped a possibly `null` object in an `Optional` class. We can manipulate the value[^2] wrapped in an `Optional` with the `map` function. The `map` method applies the given method only if the value is present, preventing us from writing a if-not-null check and the danger of `NullPointerException` if we forget to check.

Issues with Java `Optional`
Java's `Optional` is not very well-designed. It is unfortunate that Java 8 provides a `get()` method to allow retrieval of the object inside the functor, with the possibility of causing a run-time exception if `Optional` is empty. The whole point of using `Optional` is to be safe from run-time exception! Not to mentioned that Java Collections Framework does not support `Optional`.

Monad

A monad also takes in a function and returns a monad. But, unlike functor, it takes in a function that returns a monad.

```
interface Monad<T> {
  public <R> Monad<R> f(Function<T,Monad<R>> func);
}
```

The pattern above might look complicated, but you have actually seen it before:

```
interface Stream<T> {
  public <R> Stream<R> flatMap(Function<T,Stream<R>> mapper);
}
```

This interface should look familiar to you[2][#fn:3]. We have seen monads before -- `Stream` is a monad. In contrast, unless you implemented `flatMap` for `InfiniteList` or `LambdaList`, they are not monads.

Monad Laws

Just like functors, there are some laws that a monad have to follow:

- there should be an `of` operation that takes an object (or multiple objects) and wrap it/them into a monad. Further,
  - `Monad.of(x).flatMap(f)` should be equal to `f(x)` (called the *left identity* law)
  - `monad.flatMap(x -> Monad.of(x))` should be equal to `monad` (called the *right identity* law)
- the `flatMap` operation should be associative (associative law): `monad.flatMap(f).flatMap(g)` should be equal to `monad.flatMap(x -> f(x).flatMap(g))`

In other languages
The `flatMap` and `of` operations are sometimes known as the `bind` and `unit` operations respectively (e.g., in Haskell).

Knowing what is a monad is useful, since if I tell you something is a monad, you should recognize that it supports a given interface. For instance, I tell you that `Optional` is a monad. You should know that `Optional` supports the `of` and `flatMap` operation. The name of the operations may be different, but they must exists in a monad and follows the monad laws.

We won't proof formally that `Optional` follows the laws of monad in this class, but let's explore a bit more to convince ourselves that it does. Let's write the `flatMap` method for `Optional`, which is not that difficult:

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
  if (!isPresent()) {
    return empty();
  } else {
    return mapper.apply(value);
  }
}
```

Let check:

- Left identity law: `Optional.of(x).flatMap(f)` will return `f.apply(x)` (i.e., $f(1)$).
- Right identity law: `opt.flatMap(x -> Optional.of(x))` will apply `x -> Optional.of(x)` on the value of `opt`, if it exists, resulting in `Optional.of(value)`, which is `opt`. If the value does not exist (`Optional` is empty), then `flatMap` will not apply the lambda, instead it will return `empty()` right away. So it obeys the law.
- Associative law: `opt.flatMap(f).flatMap(g)` is the same as `f.apply(value).flatMap(g)`; `opt.flatMap(x -> f(x).flatMap(g))` will apply the lambda to `value`, so we get `f.apply(value).flatMap(g)`. They are the same. If `opt` is empty, then `flatMap` returns `empty` for both cases.

So, despite the complicated-sounding laws, they are actually easy to verify.

Implementing Strategy or Policy

Polymorphism is one of the pillars of OO programming. We have seen how, through inheritance and polymorphism, we can extend the behavior of a class. In some cases, polymorphism is useful for implementing different behaviors for different subclasses. For instance,

```
class Server {
  Server() { }
  abstract boolean needRest();
  abstract void available();
}

class HumanServer extends Server {
  boolean needRest() {
    return true;
  }
  void available() {
    say("Next, please!");
  }
}

class MachineServer extends Server {
  boolean needRest() {
    return false;
  }
  void available() {
    turnLightGreen();
  }
}
```

We override the method `needRest` to indicate if a particular server needs to rest, and `available` to perform an action when the server becomes available. This is sometimes known as the *strategy* pattern or the *policy* pattern, where each class encapsulates a different way of achiving the same thing.

You may see that, since `needRest` only returns a constant, one could easily store that in a field.

```
class Server {
  boolean needRest;

  Server(boolean needRest) {
    this.needRest = needRest;
  }

  boolean needRest() {
    return needRest;
  }

  abstract void available();
}

class HumanServer extends Server {
  void available() {
    say("Next, please!");
  }
}

class MachineServer extends Server {
  void available() {
    turnLightGreen();
  }
}
```

What about the strategy? With lambda expressions, it turns out that we can store the body of the method in a field as well. The `available` method takes in no argument and returns nothing, so the functional interface `Runnable` is perfect for this:

```
class Server {
  boolean needRest;
  Runnable availableAction;

  Server(boolean needRest, Runnable action) {
    this.needRest = needRest;
    this.availableAction = action;
  }

  boolean needRest() {
    return needRest;
  }

  void available() {
    availableAction.run();
  }
}
```

We have removed two subclasses. Instead of:

```
h = new HumanServer();
```

```
m = new MachineServer();
```

we can do

```
h = new Server(true,  () -> say("Next, please!"));
m = new Server(false, () -> turnLightGreen());
```

Such style of code makes the intention more explicit (no need to trace through different class files to understand the behavior), but exposes some implementation details. In terms of extensibility, it is easier, since we no longer need to create subclasses to add a new type of behavior. For example, we can say:

```
new Server(true, () -> announceNextQueueNumber());
```

Either of these style (OOP or FP) is better than having to write switch statements or if-then-else statements if we code in imperative style.

Observer Pattern

Often, in our code, an event can trigger a series of actions. For instance, pressing a button on the GUI could trigger an update to the user interface, a sound to be played, an action to be performed, etc. While one could hardcode these responses to a trigger, it would be nicer if one could add a customer action in response to a trigger. For instance, we might want to say, log an entry into a file when the button is pressed.

In OO design, this is known as the Observer pattern. With lambda, we can implement something like this with a list of lambda expressions:

```
class Event {
  List<Runnable> actions;

    Event() {
        actions = new ArrayList<>();
    }

  void register(Runnable r) {
    actions.add(r);
  }

  void trigger() {
    for (Runnable r: actions) {
      r.run();
    }
  }
}
```

This allows us to cleanly separate the different *concerns*. In the button example, which is a GUI component, we no longer need to mixed code dealing with sounds, logging, nor application-specific actions. Each of these can be implemented in a different packages, and only need to register their code to the list of actions to perform when the button is pressed.

Exercise

1. The interface `SummaryStrategy` has a single abstract method `summarize`, allowing any implementing class to define its own strategy of summarizing a long `String` to within the length of a given `lengthLimit`. The declaration of the interface is as follows:

   ```
   @FunctionalInterface
   interface SummaryStrategy {
     String summarize(String text, int lengthLimit);
   }
   ```

   There is another method `createSnippet` that takes in a `SummaryStrategy` object as an argument.

   ```
   void createSnippet(SummaryStrategy strategy) {
       :
   }
   ```

   Suppose that there is a class `TextShortener` with a static method `String shorten(String s, int n)` that shortens the String `s` to within the length of `n`. This method can serve as a summary strategy, and you want to use `shorten` as a `SummaryStrategy` in the method `createSnippet`.

   Show how you would call `createSnippet` with the static method `shorten` from the class `TextShortener` as the strategy.

2. Suppose we have a snippet of code as follows,

   ```
   Double d = foo(i);
   String s = bar(d);
   ```

   We can write it either as:

   ```
   stream.map(i -> foo(i)).map(d -> bar(d));
   ```

   or

   ```
   stream.map(i -> bar(foo(i)))
   ```

   We can be assured that the expressions above are the same because a stream is a functor. Why? Explain by indicating which law ensures the behavior above is true.

---

1. In fact, no functors in Java 8 does, since this is the interface I created just to explain the pattern of a functor.
2. Just a reminder again that these are not real interfaces in Java but just something to show you the types of input/output to a monad in a language that you are familiar with.