Interface Inheritance, Functors, and Binary Search Trees

Important Dates:

- Assigned: November 20, 2024
- Deadline: December 4, 2024 at 11:59 PM EST

Objectives:

- Students understand what it means for an interface to extend another.
- Students examine concepts from other functional programming languages, e.g., Haskell.
- Students design a class hierarchy that emulates the structure of a binary tree.

What To Do:

This problem set is an *extra credit* problem set that will add on top of your "midterm improvement" score. The score that you earn in the autograder will be normalized to a value between $[0, 5]$.

As a word of warning, you will need to spend some extra time outside of class to complete the problem set. Because we never talked about binary trees and only provide a handout on interface inheritance, a bit of self-study is required. None of this will be tested on the final exam.

As a second word of warning, this problem set is *hard*. Do not be surprised if you don't understand the majority of it the first time you look at it. Read through the entire thing, figure out what you do and don't understand, do research, and come back. This cycle may repeat. Note that its difficulty isn't due to the amount of code that you have to write.

Design classes with the given specification in each problem, along with the appropriate test suite.
Do not round your solutions!

You must write sufficient tests and adequate documentation.

Java's streams have made object compositionality a dream, at least when compared to what it was like in previous versions of Java. For example, suppose we have a `List<Optional<Integer>>` and we want to add one to every element. As a first attempt we might try the following:

```
List<Optional<Integer>> ls = // assume populated.
List<Optional<Integer>> ls2 = ls.stream().map(v -> v + 1).toList()
```

And it doesn't work because we're attempting to map a function that receives an `Optional` and adds an integer literal to it, which is non-sensical. We instead need to write the following:

```
List<Optional<Integer>> ls = // assume populated.
List<Optional<Integer>> ls2 = ls.stream()
                                .map(v -> Optional.of(v.get() + 1))
                                .toList();
```

Unfortunately, this *still* doesn't work, because what if there's an empty optional inside the list? We could filter all of those out...

```
List<Optional<Integer>> ls = // assume populated.
List<Optional<Integer>> ls2 = ls.stream()
                                .filter(o -> o.isPresent())
                                .map(v -> Optional.of(v.get() + 1))
                                .toList();
```

But at this point, we've made our code ridiculously hard to read, all to do nothing more than add one to every element in the optional type! There has to be a better way, and indeed, there is, through an operation called `fmap`. What we're after is a way of applying a function f to the *underlying data* within a type T. In this case, the type T is `Optional`, and the underlying data is an integer. Therefore, we receive a value of type T that stores an integer, a function f receives a number and returns a number, and we return a new T that wraps the value returned by f.

That seems complicated, and it is, but what's nice is that we aren't restricted to working with `Optional` types; we can make *any* type a *Functor*, meaning that it supports the `fmap` operation.

Functors are a component of a branch of very abstract math called category theory, but they also make an appearance in functional programming languages, the most prominently of those being Haskell. In Haskell, the syntax is a bit more forgiving than Java, since it (Haskell) natively supports functors.

In Java, we need to be rather clever by designing the `IFunctor` interface. The interface is generic, abstracting over a type T. The interface provides one method: `<R> IFunctor<R> fmap(Function<T, R> f)`. Don't let the signature be frightening, as all it means is: `fmap` receives a function f from type T to R, and the class returns a `IFunctor` over a new type R. The `fmap` method describes *how* we want to apply f to a particular type.

```
import java.util.function.Function;

interface IFunctor<T> {

  <R> IFunctor<R> fmap(Function<T, R> f);
}
```

The idea is that we want to designate that our optionals are a kind of functor, meaning they implement the `IFunctor` interface. Unfortunately, this *does* mean that we have to create a separate object hierarchy for an "optional" type. Because we are close to Haskell territory, let's design the generic `IMaybe` interface, which is implemented by the `Just` and `Nothing` classes.

```
interface IMaybe<T> extends IFunctor<T> {

  boolean isPresent();
  boolean isEmpty();
  T get();
}
```

Wait, interfaces can *extend* other interfaces?! Indeed so! This means that any class to implement `IMaybe`, namely `Just` and `Nothing` must override all methods in both `IMaybe` and `IFunctor`.

The `Nothing` class is incredibly straightforward—it is impossible to "fmap" a function f over nothing, so it simply returns a new instance of `Nothing`.

```
import java.util.function.Function;

class Nothing<T> implements IMaybe<T> {

  Nothing() {}

  @Override
  public <R> IFunctor<R> fmap(Function<T, R> f) {
    return new Nothing<>();
  }

  @Override
  public boolean isPresent() { return false; }

  @Override
  public boolean isEmpty() { return true; }

  @Override
```

```
  public T get() { return null; }

  @Override
  public String toString() { return "Nothing"; }
}
```

Applying "fmap" to a `Just` is only moderately more difficult: again, what's the idea behind "fmap?" We want to unwrap the object, apply f to the underlying data, then wrap it inside the object type. In this instance, we unwrap the `Just`, apply f to its value, then re-wrap it in a new instance of `Just`.

```
import java.util.function.Function;

class Just<T> implements IMaybe<T> {

  private final T DATA;

  Just(T data) { this.DATA = data; }

  @Override
  public <R> IFunctor<R> fmap(Function<T, R> f) {
    return new Just<>(f.apply(this.DATA));
  }

  @Override
  public boolean isPresent() { return true; }

  @Override
  public boolean isEmpty() { return false; }

    @Override
  public T get() { return this.DATA; }

  @Override
  public String toString() { return this.DATA.toString(); }
}
```

Now, instead of creating a `List<Optional<Integer>>`, we have to instantiate the list to be `List<IMaybe<Integer>>`, but the idea is the same. Now, we want to map, over each element of the list, the `fmap` function inside the `IMaybe`. We do not need to check whether the object is a `Nothing` or a `Just`, because that process is taken care of by the implementation of `fmap` in those classes!

```
List<IMaybe> ls = List.of(new Just<>(42), new Empty<>(), new Just(117));
LIst<IMaybe> res = ls.stream()
                     .map(m -> m.fmap(n -> n + 1))
                     .toList();
System.out.println(res); // [43, Nothing, 118]
```

What's the value in all of this hard work? We have a way of unwrapping *any* type, while maintaining its structure, to apply a function to its underlying data, then re-wrap it in the type.

In the next step, you will see how this works when applied to binary search tree nodes. The larger context is that it is absurdly cumbersome to apply a function to each node of a binary search tree, even one as simple as adding one to every element. Using functors and "fmap" makes the process unbelievably easy. Yes, five pages of background was necessary... I hope that you didn't just skip over it!

(a) First, design the `ITree<T>` interface. It should *extend* the `IFunctor<T>` interface, and provide three methods: `ITree<T> left()`, `ITree<T> right()`, and `ITree<T> value()`.

(b) Next, design two classes that implement `ITree<T>`: `Empty<T>` and `Node<T>`. The former represents an empty binary tree, and the latter represents a node in the tree that contains data. It may or may not have left and right children. When overriding `left()`, `right()`, and `value()` in `Empty`, return `null`. Hint: if you get stuck trying to implement this, I refer you to question 1 on the Fall 2023 final exam, as the class hierarchy is nearly identical!

(c) Override the `public boolean equals(Object o)` method in the `Empty` and `Node` classes. Comparing `Empty` against anything else is trivial, so we won't describe it. Comparing a `Node` against another node is more tricky. We need to first see if a node's value is the same as another node's value, and then recurse on both children. You do not need to override `public int hashCode()`.

(d) Override the `public String toString()` method in the `Empty` and `Node` classes. The string representation of an `Empty` is `"Empty"`. The string representation of a `Node` is `Node(data, left, right)`, where `data` is the value at that node, `left` is the stringified left child, and `right` is the stringified right child. Do *not* over-complicate this!

(e) When designing `Node<T>` and `Empty<T>`, you were forced to override `fmap`, but perhaps its body eludes you. Let's first consider how to "fmap" a function f over an `Empty<T>`. Doing so is absolutely trivial: we just return a new `Empty<>()` node, because there's no possible way that we can apply f to non-existent data.

The other case, namely applying "fmap" to a `Node<T>` is a bit more complex. To do so, we apply f to the data encapsulated by the `Node<T>`, then recursively apply `fmap` to its left and right children, creating a new `Node` in the process. Follow the types! Fall back on the examples we provided earlier if you get stuck.

(f) Finally, test your implementation. Again, as we stated, what's beautiful about functors is that we can apply a function over the type and maintain the structure of that type. Our test will create a binary tree and add one to each element, creating a new tree in the process. (Note that this is the canonical example of functors with tree nodes.)

```java
class ITreeTester {

  @Test
  void testITree() {
    ITree<Integer> t1 =
      new Node(5,
        new Node(1, new Empty<>(), new Empty<>()),
        new Node(3, new Empty<>(), new Empty<>()),
      new Node<>(7,
        new Node<>(6, new Empty<>(), new Empty<>()),
        new Node<>(8, new Empty<>(), new Empty<>())));

    ITree<Integer> et1 =
      new Node<>(6,
        new Node<>(4,
          new Node<>(2, new Empty<>(), new Empty<>()),
          new Node<>(5, new Empty<>(), new Empty<>())),
        new Node<>(8,
          new Node<>(7, new Empty<>(), new Empty<>()),
          new Node<>(9, new Empty<>(), new Empty<>())));

    assertEquals(et1, t1.fmap(n -> n + 1));
  }
}
```