Monads for the Java Developer

Ryan Tay, Ye Guoquan, Teoh Zhixiang

ABSTRACT: Monads are one of the core functional programming concepts, indispensable to several pure languages like Haskell and Scala, in order to preserve referential transparency and side-effect free computation. This paper explores the translation of monads in functional programming languages, particularly Haskell, to the syntax of Java. It first discusses what monads in Java entail, then delves into use cases for monads in Java, and finally closes with comparing the differences between using and not using monads in Java.

Introduction

Monad is a concept originally from category theory where it is defined as a functor with additional structure [7]. It has become more sophisticated over the years since it was introduced into Computer Science in the early 1990s. Monads are widely used in purely functional programming as they integrate some flexibility provided by imperative programming language and yet produce no side effect [14].

Monads have become popular in non-functional programming languages as well. Since the release of Java 8, the Java language supports multiple features such as Stream and Optional which are monad implementations [13]. This draws the interest of Java developers into understanding monads and provides the motivation for this paper to investigate the strength and limitation of using monads in Java.

Monads in Java

To understand monad, it is convenient to draw reference from Haskell, a purely functional programming language. In other words, the concept of monads (in functional programming) has become synonymous with Haskell's Control.Monad Typeclass[1]:

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

In our discussion that follows, we shall simplify the above definition to the following:

```
class Monad m where
  bind :: m a -> (a -> m b) -> m b
  unit :: a -> m a
```

[1] Haskell Docs on Control.Monad

In essence, the bind method binds a monad with a function that transforms a value into a monad and outputs another monad. The unit method transforms a value into monad without changing the value.

Based on the Haskell type class above, we can write a monad interface in Java with similar features.

```
public interface Monad<V> {
    static Monad<V> unit(V v);
    <R> Monad<R> bind(Function<V, Monad<R>> f);
}
```

Java 8's Functional interface and generics type system allows us to translate the above Haskell definitions to Java syntax.

Monad Laws

There are three laws that a monad must follow to avoid side effects. We will compare the Haskell example with the Java equivalent monad interface to see how the laws are abided by.

1. Right identity; i.e., binding a monad to the unit function is equivalent to the monad, since unit "lifts" or "wraps" the value (that was unwrapped implicitly in bind):

    ```
    bind m unit = m
    // in Java (== represents equivalence),
    m.bind(m::unit) = m
    ```

2. Left identity; i.e., wrap a value with context and bind it to a function (of the form $a \rightarrow m\ \ b$) is equivalent to directly applying the function on the value:

    ```
    bind (unit v) f = f v
    // in Java (== represents equivalence),
    unit(v).bind(f) = f.apply(v)
    ```

3. Associativity; i.e., the order of bind does not matter:

    ```
    bind m (\v -> bind (f v) g) = bind (bind m f) g
    // in Java (== represents equivalence),
    m.bind(f).bind(g) = m.bind((v) -> f.apply(v).bind(g))
    ```

With these definitions, any Java class that we want to implement with monadic capabilities need only define their individual implementations of these methods, as we shall see in the following sections. The conformity of the methods to the described Monad Laws is implementation-dependent, as we shall see.

Use Cases of Monads in Java

Foundation of Java Monads

The unit operator "lifts" an "unwrapped" value to a context-wrapped type. In Java syntax, this is represented as:

```
static <A> M<A> unit(A a)
```

This means "unit is a method that takes in an object of unwrapped generic type A, and returns an object of monadic context-wrapped generic type $M\langle A\rangle$".

In Haskell, functions are curried, so the bind function is actually a function that takes in one argument $m\ a$, then returns a function that takes in a function of the form ($a \rightarrow m\ b$) and returns $m\ b$. Using Java 8's Functional Interface, the bind operator is directly translated from Haskell as:

```
static <A, B> Function<Function<A, M<B>>, M<B>> bind(M<A> m)
```

However, to keep more in line with the flavor of Java syntax, we rewrite it as:

```
static <A, B> M<B> bind(M<A> m, Function<A, M<B>> f)
```

Notice how the arity of the first variant is 1, while that of the second variant is 2. With these definitions, any Java class that we want to implement with monadic capabilities need only define their individual implementations of these methods, as we shall see in the following sections. The conformity of the methods to the Monad Laws is implementation-dependent, as we shall see.

Examples of Monad Implementations in Java

Unfortunately, Java's inherent type system is not "high-level" or "strong" enough to support a Haskell-like Typeclass implementation of monads, via Java interfaces. We will first explore some examples of monad implementations in Java, and their syntax and implications, to gain insight into the flavour of functional programming in Java. Then, we will use this insight to discuss the limitations of Java's type system in implementing a generic monad interface similar to Haskell's Monad Typeclass.

Maybe Class

The Optional monad in Java aims to resemble Haskell's Maybe monad instance. As a result, we include Haskell's Maybe monad as our reference. From Haskell Docs, the Maybe type "is a simple kind of error monad, where all errors are represented by Nothing." In short, the Maybe type is a sum type of two possible values, Just and Nothing. The instance definition of Haskell's Maybe is:

```
instance Monad Maybe where
    (Just x) >>= k = k x
    Nothing  >>= _ = Nothing
```

Notice that the unit (or return) method is implicit here; it is assumed to simply be of the form $(x \rightarrow Maybe\ x)$. Translating this to align with our previous definitions of unit and bind:

```
instance Monad Maybe where
    unit x          = Maybe x
    bind (Just x) k = k x
    bind Nothing  _ = Nothing
```

We shall first endeavour to define a similar Java class, Maybe, that implements these methods.

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

class Maybe<A> {
    private A a;

    private Maybe(A aVal) {
```

```
        if (!isNull(aVal)) a = aVal;
        else a = null;
    }
    static <A> Maybe<A> unit(A a) {
        return new Maybe<>(a);
    }
    <B> Maybe<B> bind(Function<A, Maybe<B>> f) {
        if (!isNull(a)) return f.apply(a);
        else return new Maybe<B>(null);
    }
    boolean isNull(A a) {
        return a == null;
    }
    public String toString() {
        if (!isNull(a)) return "Just " + a;
        else return "Nothing";
    }
    public static void main(String[] args) {
        Maybe<Integer> a = Maybe.unit(3);
        Maybe<Integer> b = Maybe.unit(null);

        Maybe<Integer> sum =
            a.bind(val1 ->
            b.bind(val2 ->
            Maybe.unit(val1 + val2)
        ));

        System.out.println(sum); // Nothing
    }
}
```

A few things to note in the above implementation:

- The return type and parameters of bind are changed to reflect an instance method implementation, with the original first monad parameter being implicitly represented by the monad object instance
- sum is the result of a series of chained bind operations
- The arguments $Function\langle A, Maybe\langle B\rangle\rangle\ f$ here are implicitly defined via Java 8's lambda notation, as in $(val1 \rightarrow \ldots)$

The above implementation proves to us that it is entirely possible to create Java synonyms of select monad types, and with a modest amount of code. The Maybe type is so useful and ubiquitous that other languages, like Swift[2], now have synonymous implementations. In Java, like Swift, the synonymous monad class is the Optional[3] [5] class, with these key method implementations:

```
public T get() {
    if (value == null) throw new NoSuchElementException("No value present");
    return value;
}
public T orElse(T other) {
```

[2] Swift's Optional defines a bind operation, and is known for its "Optional chaining" capabilities

[3] For explanations or discussion on super and extends, see [4].

```
    return value != null ? value : other;
}
public static <T> Optional<T> ofNullable(T value) {
    return value == null ? empty() : of(value);
}
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent()) return empty();
    else return Optional.ofNullable(mapper.apply(value));
}
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent()) return empty();
    else return Objects.requireNonNull(mapper.apply(value));
}
```

Here, ofNullable corresponds to unit, and flatMap corresponds to bind. Nothing is represented by null, and Just is represented by simply the object's non-null value. The next natural question to answer is whether the Optional class obeys the Monad Laws (i.e., monoid properties). To investigate this, we test [11] the left-identity, right-identity, and associativity, of the flatMap (bind) operator.

Testing left-identity:

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

public class OptionalTestLeftIdentity {
    public static void main(String[] args) {
        Function<Integer, Optional<Integer>> f = (x -> {
            if (x == null) return Optional.ofNullable(-1);
            else return Optional.ofNullable(null);
        });
        System.out.println(
            Optional.ofNullable((Integer) null).flatMap(f).equals(f.apply(null))
        ); // false
    }
}
```

The problem[4] here arises only when a null value is both passed to the left-hand and right-hand sides. The idea is that the left-hand side immediately returns an Optional.empty() object as defined in the source, while the right-hand side results in the creation of a new Optional object as defined by the Function f. This result has significant implications on the use of Java's Optional class as part of one's functional programming paradigm, since referential transparency can be threatened.

Testing right-identity:

```
import java.util.Optional;

public class OptionalTestRightIdentity {
    public static void main(String[] args) {
```

[4] It might be interesting to note that there are arguments for the contrary (that Java's Optional preserves left-identity) that have been rightly debunked.

```
        Optional<Integer> m = Optional.ofNullable(null);
        System.out.println(m.flatMap(Optional::ofNullable).equals(m)); // false
    }
}
```

On the other hand, right-identity is preserved because it does not deal with nor depend on the implementation of the Function f. Lastly, we test associativity of flatMap:

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

public class OptionalTestAssociativity {
    public static void main(String[] args) {
        Function<Integer, Optional<Integer>> f = x -> {
            (x % 2 == 0) ? Optional.ofNullable(null)
            : Optional.ofNullable(x);
        }
        Function<Integer, Optional<Integer>> g = y -> {
            (y == null)  ? Optional.ofNullable(null)
            : Optional.ofNullable(y);
        }
        Optional<Integer> m = Optional.of(2);
        Optional<Integer> lhs = m.flatMap(f).flatMap(g);
        Optional<Integer> rhs = m.flatMap(y -> f.apply(y).flatMap(g));
        System.out.println(lhs.equals(rhs); // true
    }
}
```

Crucially, while map[5] might not preserve the Monad Law of associativity, flatMap[6] does.

Having built a functional version of the Maybe monad in Java, and then verifying the validity of the Java built-in monadic Optional class, it is apparent that the Optional monad has several benefits over more orthodox imperative null-checking that we are used to, namely in minimising boilerplate, increasing readability and maintainability, encapsulating implementation details (in this case of error-checking), and composing functions. With implementing a monad class, we get most of the benefits that functional programming provides while using the class, such as referential transparency[7] and side-effect-free implementation[8].

Either Class

In Java, error-handling is most commonly done via throwing exceptions—for instance, when erroneous input is received that produces errors in method evaluation. In Haskell, the Either type is defined on two values, Left a and Right b, as in Either a b, with Left usually representing an error, and Right representing successful evaluation. In a similar flavour to Optional, one can devise a Java implementation [1] of Haskell's Either monad:

---

[5] Covered in this article.

[6] Also shown in this code snippet.

[7] As we have seen above, referential transparency breaks when one uses map() or encounters the failing scenario of left-identity.

[8] With regards to Java's Optional class, not using map() is one of the ways of achieving this.

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

public class Either<E, A> {
    private Optional<E> error;
    private Optional<A> a;

    private Either(E error, A a) {
        this.error = Optional.ofNullable(error);
        this.a     = Optional.ofNullable(a);
    }
    public static <E, A> Either<E, A> unit(E error, A a) {
         if (error != null) return new Either<>(error, null);
         else return new Either<>(null, a);
    }
    public <B> Either<E, B> bind(Function<A, Either<E, B>> f) {
         if (!isError()) return f.apply(a.get());
         else return Either.unit(error.get(), (B) null);
    }
    boolean isError() {
         return error.isPresent();
    }
    public String toString() {
        if (!error.isPresent()) return "Right " + a.get();
         else return "Left " + error.get();
    }
}
```

Notice the following:

- The two fields a and error are Optional-typed objects
- The get() method in bind's implementation is from Optional::get, which evidently has public visibility. This means that it has the potential to violate side-effect-free computation, especially if one uses it in this manner:

```
Optional<Integer> m = Optional.of(3);
// The purpose of wrapping it in a monadic context is to minimise side effects.
// However, this is not pure:
int i = m.get() + 5; // i = 8
```

The Either class can be used in this manner:

```
public class EitherTest {
    public static void main(String[] args) {
        Either<Exception, Integer> a = either(3);
        Either<Exception, Integer> b = either(5);
         Either<Exception, Integer> c = either(null);
         Either<Exception, Integer> sum =
              a.bind(e1 ->
              b.bind(e2 ->
              either(e1 + e2)
         ));
         Either<Exception, Integer> sum1 =
              a.bind(e1 ->
              c.bind(e2 ->
              either(e1 + e2)
```

```
        ));
        System.out.println(sum);  // Right 8
        System.out.println(sum1); // Left java.lang.NumberFormatException
    }
    private static Either<Exception, Integer> either(Integer value) {
        if (value == null)
            return Either.unit(new NumberFormatException(), null);
        else return Either.unit(null, value);
    }
}
```

Notice how in the above code, no explicit error-checking was performed, except in either() (necessary for the simplicity of the code snippet), since the error-checking has been encapsulated in the Either type, as it should be! Also, no matter the order of calling bind, the Monad Laws ensure that the same result is produced, in an almost[9] fully referential transparent manner.

Limitations of Java Monads

By defining monads in Java by way of individual classes, we have implemented instances of monads, but not the monad type. In order to achieve the latter, we must devise a generic higher-order interface that can describe the rules that all monad instances must follow. In Haskell lingo, this refers to the Monad Typeclass[10]. To achieve this in Java, we must define a "TypeInterface" [6] as such:

```
TypeInterface Monad<M> {
    static <A>    M<A> unit(A a);
    static <A, B> M<B> bind(M<A> m, Function<A, M<B>> f);
}
```

Notice that these method definitions were alluded to in 3.2, when devising Java analogues to Haskell's monadic return and (>>=). The key idea is that because Java's type system is built on classes and parametric polymorphism, with its generics type system and "highest-order" type being at the class level, it is really only capable of defining monad instances, and not the generic monad type that is present in Haskell and other functional languages. In Haskell, m is defined as a type that implements the two main operations return and (>>=). In the Java analogue, $M\langle A\rangle$ is defined as a class M parametrized with type A. It is important to note here that while describing m in Haskell as a type makes sense, in Java it is not right to describe M as a type analogous to Haskell's m.

Eric Lippert describes a Java class as "a name for a set of values that have something in common, such that any one of those values can be used when an instance of the class is required." In other words a class is just a set of all "like" instances. In contrast, a class in Haskell "is not a kind of type", but "describes a set of types". In short, in Java, a class is higher than an instance (object), so Java has "two levels"[11], while in Haskell, a class is higher than a type, which is higher than an instance, so Haskell has "three levels". This is why the M type that was used in the "TypeInterface" definition

[9] Once again, refer to the Optional class section's discussion on violation of left-identity.

[10] This is defined in 3.2.

[11] In Eric Lippert's response [6], he mentions how Java's generic type system actually adds a layer to the Java type hierarchy, but also that Haskell would then have four layers.

above cannot exist in Java, precisely because there is no “TypeInterface" Typeclass that is higher-order than the $M\langle A\rangle$ class.

Benefits when using monads

With monads, we do not have to write plenty of boilerplate code just to deal with cases that might cause exceptions on your functions. Monads can also help us to retrieve values as an Optional type so we do not need to write complicated if statements to ensure the values retrieved is as expected.

Examples of using monads to reduce code length

In this example provided by Castaño, it shows how lengthy the code can be with just Optional monads.

```
private abstract class Counter {
    abstract Optional<Integer> colour();
}
private abstract Optional<Counter> fetchThisMonthCounter();
private abstract Optional<Counter> fetchPreviousMonthCounter();
public Optional<Integer> totalColourCount() {
    Optional<Counter> thisMonth = fetchThisMonthCounter();
    Optional<Counter> previousMonth = fetchPreviousMonthCounter();
    if(thisMonth.isPresent() && thisMonth.get().colour().isPresent()
       && previousMonth.isPresent() && previousMonth.get().colour().isPresent())
    {
        Integer colour1 = thisMonth.get().colour().get();
        Integer colour2 = previousMonth.get().colour().get();
        return Optional.of(colour1 + colour2);
    }
    return Optional.empty();
}
```

Bringing focus to the if statement, it is just there to make sure there are values in thisMonth and previousMonth. Although there is a get() function, it will not throw an exception because the if statement will first check if there is a value in thisMonth before retrieving the value. This shows that Optionals do remove the need for exception handling.

```
public Optional<Integer> totalColour() {
   return fetchThisMonthCounts().flatMap(Counter::colour).flatMap(colour1 ->
          fetchPreviousMonthCounts().flatMap(Counter::colour).flatMap(colour2 ->
          Optional.of(colour1 + colour2)
   ));
}
```

To take it one step further, the whole function can be compressed by using flatMap to apply functions that also return an Optional. flatMap will unwrap the Optional in itself, apply the mapper function, and return the result as an Optional. If the functions fetchThisMonthCounts() or fetchPreviousMonthCounts() return an empty Optional, flatMap will return an empty Optional. This lets you skip writing multiple checks to make sure the values you get are present before performing operations on the variables.

Because of this, monads can make your code shorter and look more readable for other programmers using your code.

References

1. Andrés Castaño. 2016. Monads for Java developers, Part 2. (November 2016). Retrieved November 6, 2020, from https://medium.com/@afcastano/monads-for-java-developers-part-2-the-result-and-log-monads-a9ecc0f231bb
2. Andrés Castaño. 2016. Monads for the Java Developer Video. (3 November 2016). Retrieved November 6, 2020 from https://www.youtube.com/watch?v=vePeILeSv4E&ab_channel=MelbJVM
3. Angelika Langer. 2018. Java Generics FAQs - Type Arguments, FAQ 103. (August 2018). Retrieved November 6, 2020 from http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#FAQ103
4. Bert Fernandez. 2010. Top-voted response to “Difference between <? super T> and <? extends T> in Java". (December 2010). Retrieved November 6, 2020 from https://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java
5. David Katleman. 2014. OpenJDK Optional Class source code. (March 2014). Retrieved November 6, 2020 from http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Optional.java
6. Eric Lippert. 2016. Top-voted response to "Why can the Monad interface not be declared in Java?". (March 2016). Retrieved November 6, 2020 from https://stackoverflow.com/questions/35951818/why-can-the-monad-interface-not-be-declared-in-java
7. Eugenio Moggi. 1991. Notions of computation and monads. (.n.d.). Retrieved November 12, 2020, from https://person.dibris.unige.it/moggi-eugenio/ftp/ic91.pdf
8. Jörg Rathlev. 2015. The IO monad (or something similar) in Java. (June 2015). Retrieved November 6, 2020 from https://gist.github.com/joergrathlev/f17092d3470dcf732be6#file-demo-java-L23
9. Joshua Bloch. 2009. Effective Java - Still Effective After All These Years. pp 5-14. (October 2009). Retrieved November 6, 2020 https://www.oracle.com/technetwork/server-storage/ts-5217-159242.pdf
10. Oleg Šelajev. 2015. Unlocking the Magic of Monads in Java 8. (2 June 2015). Retrieved November 6, 2020 from https://www.youtube.com/watch?v=nkUafcNWiQE&ab_channel=OracleDevelopers
11. Marc Siegel. 2013. Does JDK8’s Optional class satisfy the Monad laws? Yes, it does. (Nov 2013). Retrieved November 6, 2020 from https://gist.github.com/ms-tg/7420496
12. Marcello La Rocca. 2016. How Optional Breaks the Monad Laws and Why It Matters. (September 2016). Retrieved November 6, 2020 from https://www.sitepoint.com/how-optional-breaks-the-monad-laws-and-why-it-matters/
13. Thomas Andolf. 2019. Write a monad, in Java, seriously? (28 May, 2019). Retrieved November 12, 2020, from https://medium.com/swlh/write-a-monad-in-java-seriously-50a9047c9839
14. Vorashil Farzaliyev. 2020. The Monad Design Pattern in Java. What is Monad? (13 March 2020). Retrieved November 12, 2020, from https://medium.com/thg-tech-blog/monad-design-pattern-in-java-3391d4095b3f