Monads
Reminder: composing Streams

• Old imperative code:

```java
List<Student> students = new ArrayList<>();
for (Country c : getAllCountries()) {
    for (University u : c.getUniversities()) {
        for (Faculty f : u.getFaculties()) {
            for (Student s : f.getStudents()) {
                students.add(s);
            }
        }
    }
}
```

• New Stream API code:

```java
List<Student> students = getAllCountries().stream()
    .flatMap(c -> c.getUniversities().stream())
    .flatMap(u -> u.getFaculties().stream())
    .flatMap(f -> f.getStudents().stream())
    .collect(Collectors.toList());
```
Reminder II: composing Optionals

• Old imperative code:

```java
// working with nulls
Student s = getStudent();
if (s != null) {
    Course c = s.getCourse("Software Design");
    if (c != null) {
        Exam e = c.getMoedA();
        if (e != null) {
            return e.getGrade();
        }
    }
}
return null;
```

• New Optional API code (assuming everything returned an Optional instead of nullable):

```java
return getStudent()
    .flatMap(x -> x.getCourse("Software Design"))
    .flatMap(Course::getMoedA)
    .flatMap(Moed::getGrade);
```
flatMap in detail

Why do we need flatMap?

- If we used map, we would get a Stream of Stream:

  ```java
  Stream<Stream<Faculties>> = univerisities.stream()
  .map(u -> u.getFaculties().stream());
  ```

  - map doesn't compose

- flatMap performs a map operation, followed by a "flatten" operation, converting a Stream<Stream<S>> into Stream<S>:

  ```java
  <S> Stream<S> flatMap(Function<T, Stream<S>> f);
  ```

  - Composes easily

- Both Stream and Optional are something called a Monad
What is a Monad?

- A monad is any class that exposes the following API:

```java
class MyMonad<T> {
    public static <T> MyMonad<T> of(T t);
    public <S> MyMonad<S> flatMap(Function<T, MyMonad<S>> f);
}
```

- Unlike other design patterns we saw, monads don't reduce coupling, increase cohesion, or use polymorphism (in Java)

- Rather, Monads give us:
  - **Shorter** and more **declarative** code
  - A **uniform** way of **composing** functionality
  - **Generic** containers that provide additional **explicit context**
  - The ability to focus on the **happy path** while taking care of the unhappy paths
The Monad laws

A proper monad has to satisfy the three monadic laws:

I. Left Identity:
   \[ \text{MyMonad.of(x).flatMap(f)} \equiv f.\text{apply(x)} \]

II. Right Identity:
    \[ m.\text{flatMap}(x \to \text{MyMonad.of(x)}) \equiv m \]

III. Associativity:
    \[ m.\text{flatMap}(f).\text{flatMap}(g) \equiv m.\text{flatMap}(x \to f.\text{apply(x)}.\text{flatMap}(g)) \]
The Box monad

The simplest monad is just a box:

```java
class Box<T> {
    private final T t;
    private Box(T t) {
        this.t = t;
    }
    public static <T> Box<T> of(T t) {
        return new Box(t);
    }
    public <S> Box<S> flatMap(Function<T, Box<S>> f) {
        return f.apply(t);
    }
}
```

• While obviously not very useful, most monads are actually a box with a few additional features
  • Optional is a box that can be empty
  • Stream is a box that can contain multiple elements
Optional: Implementation

Example implementation of Optional:

```java
class Optional<T> {
    private final T element;
    private Optional(T element) {
        this.element = element;
    }
    // monadic methods
    public static <T> Optional<T> of(T t) {
        if (t == null)
            throw new IllegalArgumentException("Use Optional.empty()");
        return new Optional(t);
    }
    public <S> Optional<S> flatMap(Function<T, Optional<S>> f) {
        return element == null ? this : f.apply(element);
    }
    // not part of the monadic interface
    public static <T> Optional<T> empty() {
        return new Optional<T>(null);
    }
}
```
map and forEach

• What about map, forEach, and other higher-order functions?
• flatMap and of is enough to implement these and others:

```java
class MyMonad<T> {
    public <S> MyMonad<S> map(Function<T, S> f) {
        return flatMap(t -> of(f.apply(t)));
    }
    public MyMonad<T> forEach(Consumer<T> f) {
        return map(t -> { f.accept(t); return t; });
    }
}
```

• map and forEach let us focus on the happy path, while taking care of the ugly details inside the monad
The Future monad: Motivation

• Suppose we have a task that takes a very long time:

```java
List<Picture> getAlbum(String name) {
    // fetches data from remote server...
    Profile p = getProfile(name);
    List<Album> as = getAlbums(p); // fetches data ...
    Album a = find(as, "Thailand"); // fetches data ...
    return getPictures(a); // fetches data ...
}
```

• How should handle this?
  • Blocking until all tasks are complete is sometimes unacceptable
  • For example, in a user interface
Future: Alternatives I

• Running requests in their own thread:

```java
new Thread() -> {
    Profile p = getProfile(name);
    List<Album> as = getAlbums(p);
    Album a = find(as, "Thailand");
    return getPictures(a);
}
```

• Problems?
  • Doesn't compose
  • Handling threads is annoying
    • Not declarative enough
  • Hard to wait for a return value
Future: Alternatives II

• The standard way of handling asynchronisity is to use callbacks:

```java
void getAlbum(String name, Consumer<List<Picture>> callback) {
    new Thread(() -> callback.accept(getAlbum(name))).start();
}
```

• Problems?

  • Still annoying to **compose** (though less so than threads)

```java
callback1(params1, t1 ->
    callback2(params2, t2 ->
        callback3(params3, t3 -> ...)
    // this is often called the pyramid of doom..
);)
```

  • Still hard to get a return value

• When we want to **compose** containers/abstractions of **generic** types, we want Monads!
Future: Class definition

- We want a generic container for a promise of an element
  - That is, a box that will, sometime in the future, contain an object
- We call this container **Future**

```java
class Future<T> {
    /* monad functions */
    static <T> Future<T> of(T t) { ... }
    <S> Future<S> flatMap(Function<T, Future<S>> f) { ... }
    /* monadic extensions */
    <S> Future<S> map(Function<T, S> f) { ... }
    Future<T> forEach(Consumer<T> c) { ... }
    /* non-monadic functions */
    // lazily creates a future on a new thread
    static <T> Future<T> of(Supplier<T> s) { ... }
    // blocks until a value is available (avoid if possible)
    T get() throws InterruptedException { ... }
}
```
Future: Solution

• Replace old code with future-aware code

```java
// old code
Profile getProfile() { ... }
// Future-aware code
Future<Profile> getProfile() { ... }
// same for all the other functions
```

• Since Future is a monad, composition is trivial

```java
Future<List<Picture>> getThailandAlbum(String userName) {
    return getProfile(name)
        .flatMap(profile -> getAlbums(profile))
        .flatMap(albums -> find(albums, "Thailand"))
        .flatMap(album -> getPictures(album));
}
```
Future: Callbacks and lifts

What about a callback?

- That's exactly the default `forEach` implementation
- Reminder:

```java
public Future<T> forEach(Consumer<T> f) {
    map(t -> {f.accept(t); t;});
}
```

- Usage in Future:

```java
getThailandAlbum("John smith")
    .forEach(GUI::displayPictures);
```

- We "lift" non-monadic functions/methods by using `map`:

```java
class Album {
    DateTime getCreationTime() { ... }
}

Future<Album> album = getAlbum(profile);
Future<DateTime> creationTime =
    album.map(Album::getCreationTime);
```
Future:

CompletableFuture\(<T>\)

• The standard library's monadic future implementation is `CompletableFuture`
  • Not to be confused with `FutureTask`, which is an old (1.5) class that is not monadic
  • Both futures implement `Future`, which is also not monadic
• Unlike `Stream` and `Option`, which use the standard monad names, it has its own unique names:
  • `of` => `completableFuture`
  • `flatMap` => `thenCompose`
  • `map` => `thenApply`
  • `forEach` => `thenAccept`
  • Many others
Monad diagram

- We prefer to program in the **happy path** (bottom row in diagram)
- But we can't due to additional concerns
  - Function might fail, take a lot of time to compute, etc.
  - We have to program under the monadic context (top row)
- Monads **wrap** our elements with classes that address these concerns
Other monads

There are other kinds of interesting monads:

• Virtually any `Collection` can be adapted to a monad
  • For some pathological cases, `Set` isn't a monad, i.e., doesn't respect the three monad laws

• `Observer<T>` can also be monadic
  • For an example implementation, see `RxJava`
  • Future is the promise of a single element; Observer is the promise of many (or zero) elements
  • Same relationship as `Optional` vs. `Stream`

• `IO` is used in pure languages like Haskell to abstract side-effects and IO operations (user input, reading files, etc.)

• `Try` is used in languages without checked exception to abstract errors
  • More general than `Optional`
  • Can also be used in Java since it's more composable than checked exceptions
Monad: Applicability

Use Monads when you want to:

• Wrap your types with an explicit context/caveat
  • While still being able to use existing functions with `map` or expose the wrapped element with `forEach`

• Write more declarative code
  • We focus on the happy path, while taking care of any additional details inside the monad

• Provide a uniform API for generic composition
Monad: Implications

✓ Easy to compose
✓ Context and other assumptions are explicit
  ✓ Types are better than comments
X "Scary" for beginners
  ✓ But what they don't know won't hurt them
X Java's type system forces us to perform a lot of code duplication
  X Can't write generic code that works on any monad
  X Have to re-implement map, forEach for every monad
✓ But you probably don't want to use the default implementation anyway
Further Reading

- [wiki.haskell.org/All_About_Monads](wiki.haskell.org/All_About_Monads) (Haskell)
- [learnyouahaskell.com/a-fistful-of-monads](learnyouahaskell.com/a-fistful-of-monads) (Haskell)
- [www.coursera.org/course/reactive](www.coursera.org/course/reactive) (Scala)
- [github.com/ReactiveX/RxJava](github.com/ReactiveX/RxJava) (Java)
- [http://blog.plover.com/prog/burritos.html](http://blog.plover.com/prog/burritos.html) (Burritos)
Implementations, additive monads, fluent APIs and further reading

APPENDIX
Fluent APIs

• In object-oriented languages, monadic transformations are **chained in the order** that they are evaluated:

```java
int sum = stream
    .map(x -> x * x)
    .filter(x -> x % 3 == 0)
    .map(x -> x + 8)
    .sum();
```

• Examples include Java’s Stream, C# LINQ (with IEnumerable), Ruby’s Enumerable module, and even bash piping.

• Without fluent APIs, we have to read and invoke functions in **reverse order**:

```java
int sum = sum(
    map(x -> x + 8,
        filter(x -> x % 3 == 0,
            map(x -> x * x, list))));
```

• Languages with non-fluent API: ML, Haskell, Python, C++, Lisp(s)
Fluent APIs (cont.)

- Often enough, any API that returns `this` from method calls is called Fluent
- This pattern is especially common with `builder` objects
- But unlike monads, it doesn’t reverse the order of reading:

```java
new Builder()
  .setX(x)
  .setY(y)
  .build();
```

- vs.

```java
Builder b = new Builder()
b.setX(x)
b.setY(y)
b.build();
```

- It’s possible to transform non-fluent APIs to fluent APIs either with wrapping / decorators, or special functions (e.g., in Haskell/F#/ML: `x |> f = f x`)
Monads in Haskell/Scala

- In languages with a richer type-system than Java, like Haskell or Scala, Monads are much more interesting/useful.
- This is because you can write code that works on a general Monad.
- For example, in our java implementation we have to implement `map`, `forEach`, `filter`, etc. for every new Monad class we define.
- If we had higher kinded types, we could implement it just once for all monads.