Building Containers for Side Effects

Esteban Herrera
JAVA ARCHITECT

@eh3rrera   www.eherrera.net

Monad

A computational context in which functions can be safely executed.

A Monad

M<T>

Is a parameterized type

T -> M<T>

Has a unit function to put a value inside of it

M<T>
bind
T -> M<U>
=
M<U>

Has a bind function to apply a function to transform the value

Optional

✓ A parameterized type: Optional<T>

✓ Unit: Optional.of()

✓ Bind: Optional.flatMap()

flatMap

```
myIntegerMonad
      .flatMap( Integer i -> i / 2.0 )
      .flatMap( Double d -> new BigDecimal(d) )
      .flatMap( /* ... */ )
```

Law of Monads #1: Associativity

```
Monad.of(value)
   .flatMap(f)
   .flatMap(g)
   .equals(
       monad.flatMap( x -> f.apply(x).flatMap(g) )
   )
```

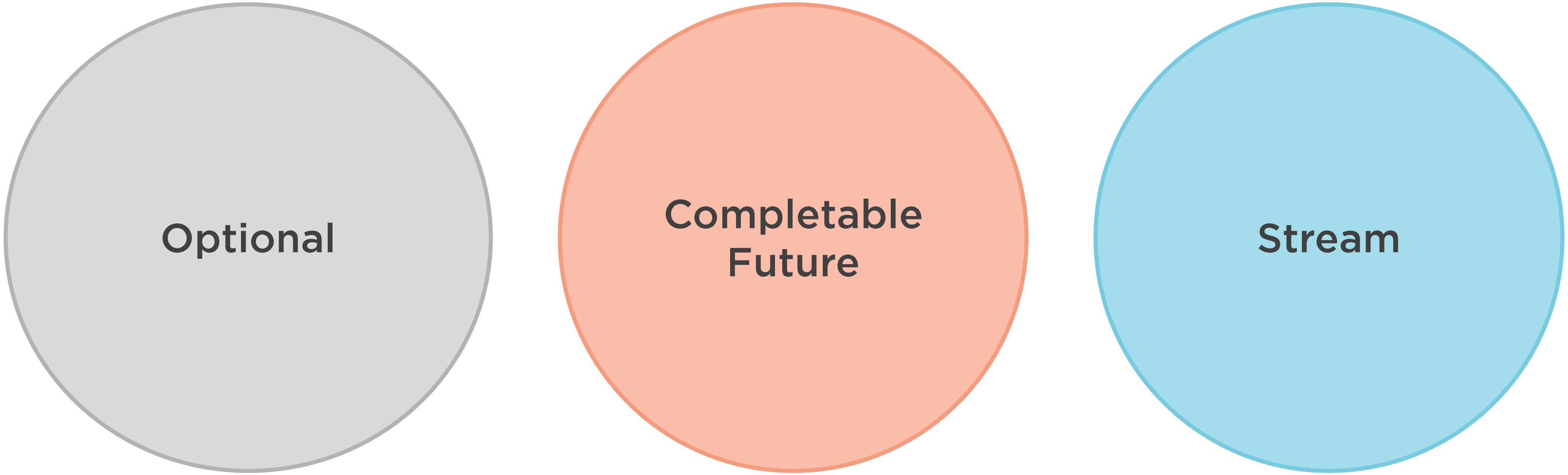
Law of Monads #2: Left Identity

```
Monad.of(value)
   .flatMap(f)
   .equals(
      f.apply(value)
   )
```

Law of Monads #3: Right Identity

```
Monad.of(value)
   .flatMap(x -> Monad.of(x))
   .equals(
      Monad.of(x)
   )
```

Monadic Types in Java

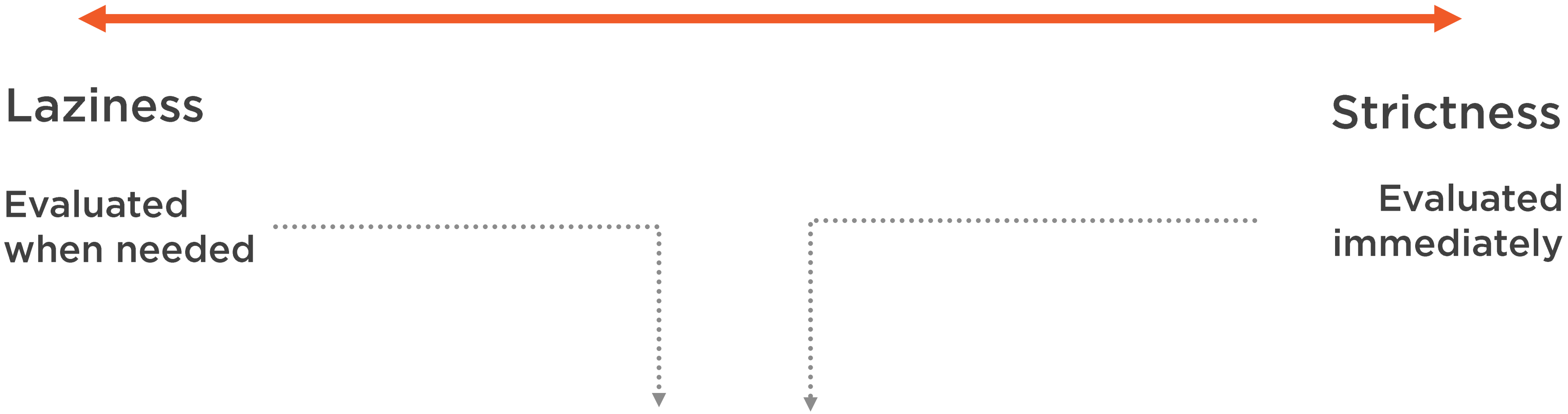


Optional

Completable Future

Stream

Understanding Laziness



Laziness

Evaluated
when needed

Strictness

Evaluated
immediately

```
String s = getString();

// ...

String getString() {
   return "Hello" + " " + "World";
}
```

This Can Be Considered Lazy

```
for (int i = 0;; i++) {
   if (i > 1_000_000) break;
}
```

This Can Be Considered Lazy

```
true || false
```

Strictness is about doing something, while laziness is about indicating that we may do something sometime in the future.

Implementing Laziness

```
public interface Supplier<T> {
    T get();
}
```

Implementing Laziness

```
public interface Consumer<T> {
    void accept(T t);
}
```

Implementing Laziness

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

Implementing Laziness

```
public interface Effect {
    void run();
}
```

```
String s = getString();
```

```
Supplier<String> s = () -> getString();

System.out.println(s.get());
```

```
Supplier<String> s = () -> getString();

Effect e = () -> System.out.println(s.get());
```

```java
public static void main(String args[]) {
   String s = getString();
   System.out.println(s);
}
```

```java
public static void main(String args[]) {
   Supplier<String> s = () -> getString();
   Effect e = () -> System.out.println(s.get());
}
```

```java
public static void main(String args[]) {
   String s = getString();
   System.out.println(s);
}
```

```java
public static void main(String args[]) {
   Supplier<String> s = () -> getString();
   Effect e = () -> System.out.println(s.get());
   e.run();
}
```

Handling Side Effects in a Functional Way

Effect

Anything that can be observed from outside the program by a user or another program.

Side Effect

Anything, besides the value returned by the function, that is observable from outside the function.

f

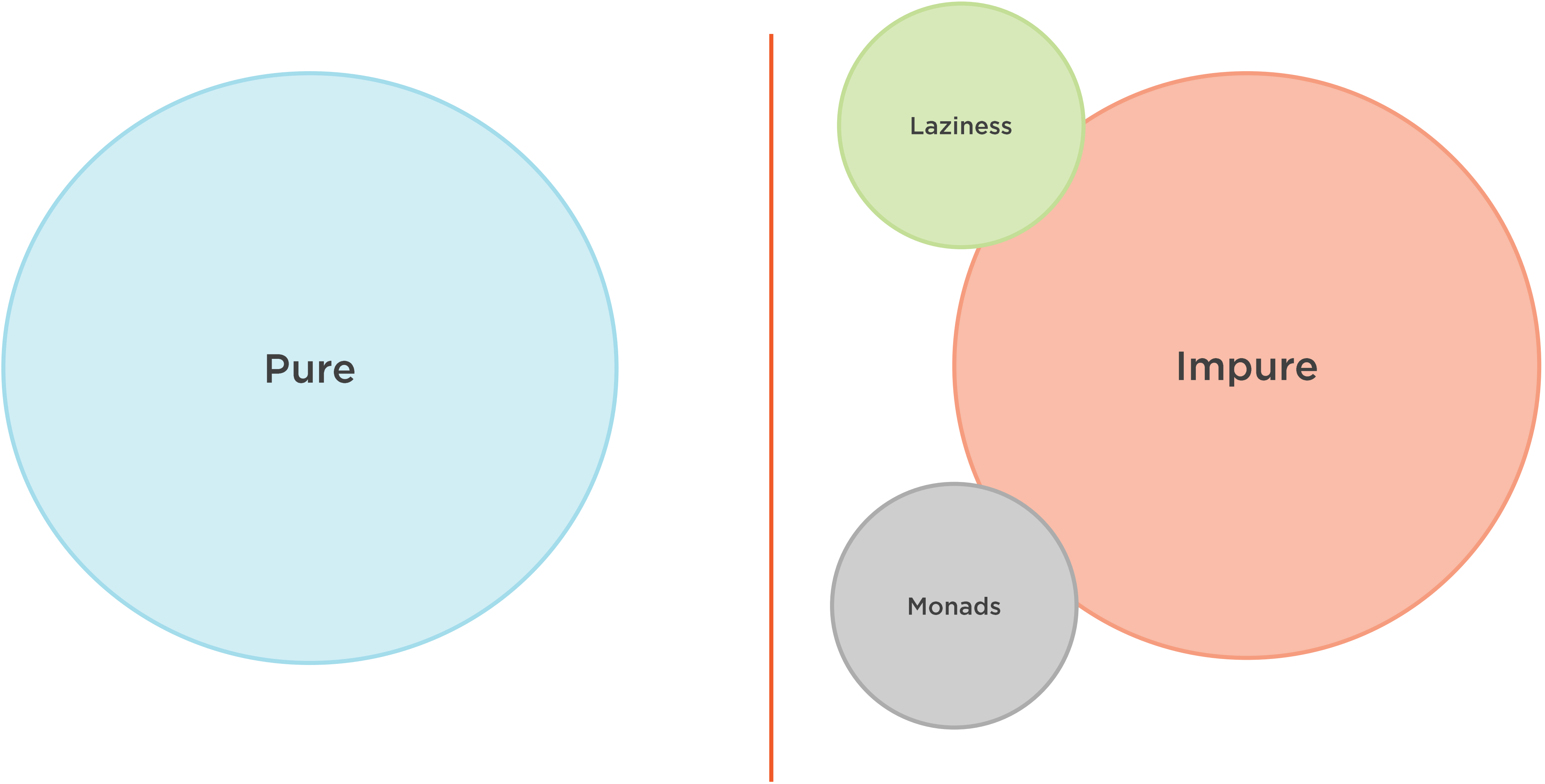
Effect

```
String x = read();
```

How do we make effects functional?

We don’t, there’s no way to do it.

Separating Pure and Impure Parts



```
String x = read();
```

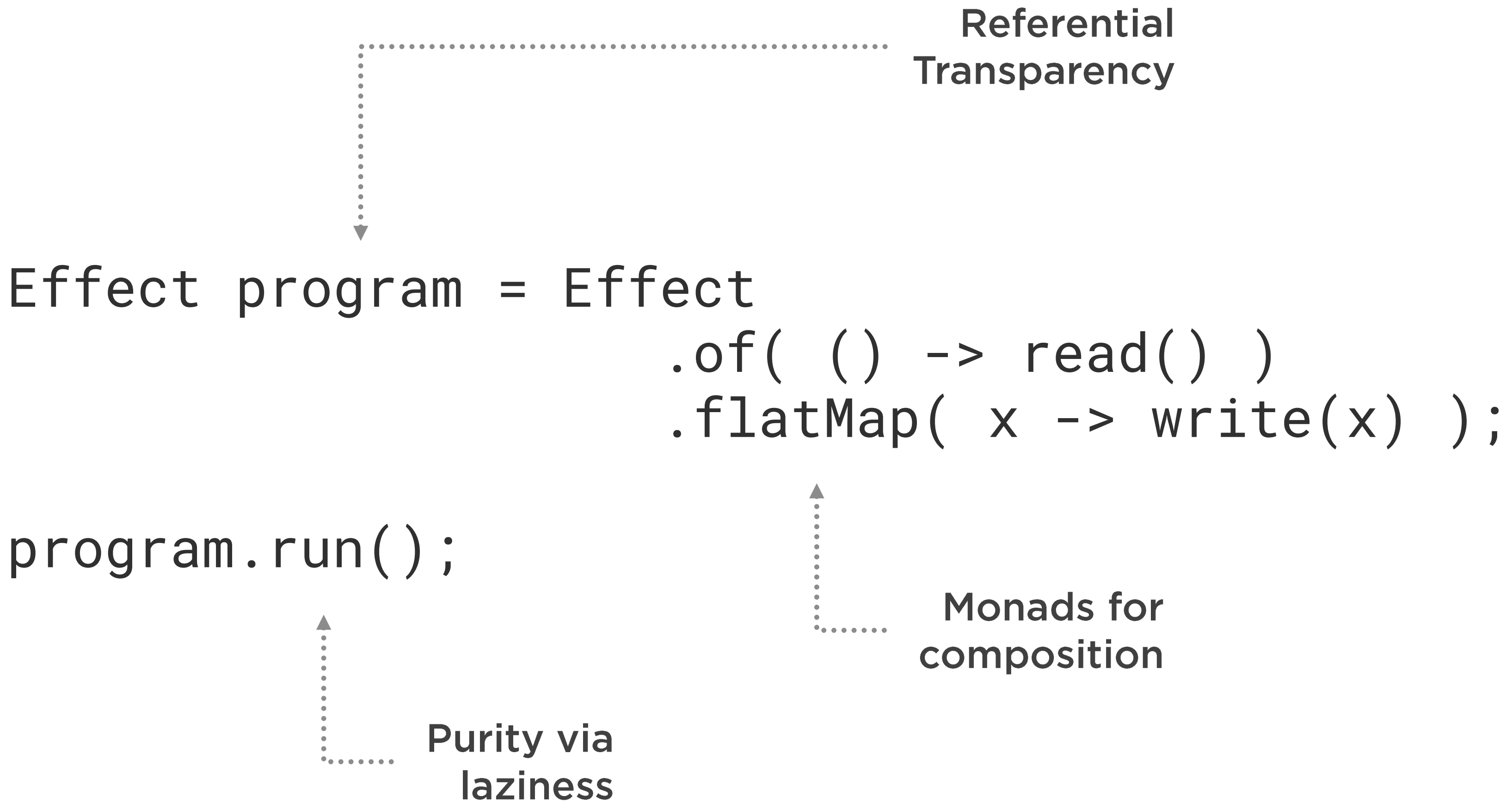
```
Effect x = read();
```

```
read() -> Effect
```

Read from keyboard

```
write(x) -> Effect
```

Write x to the screen



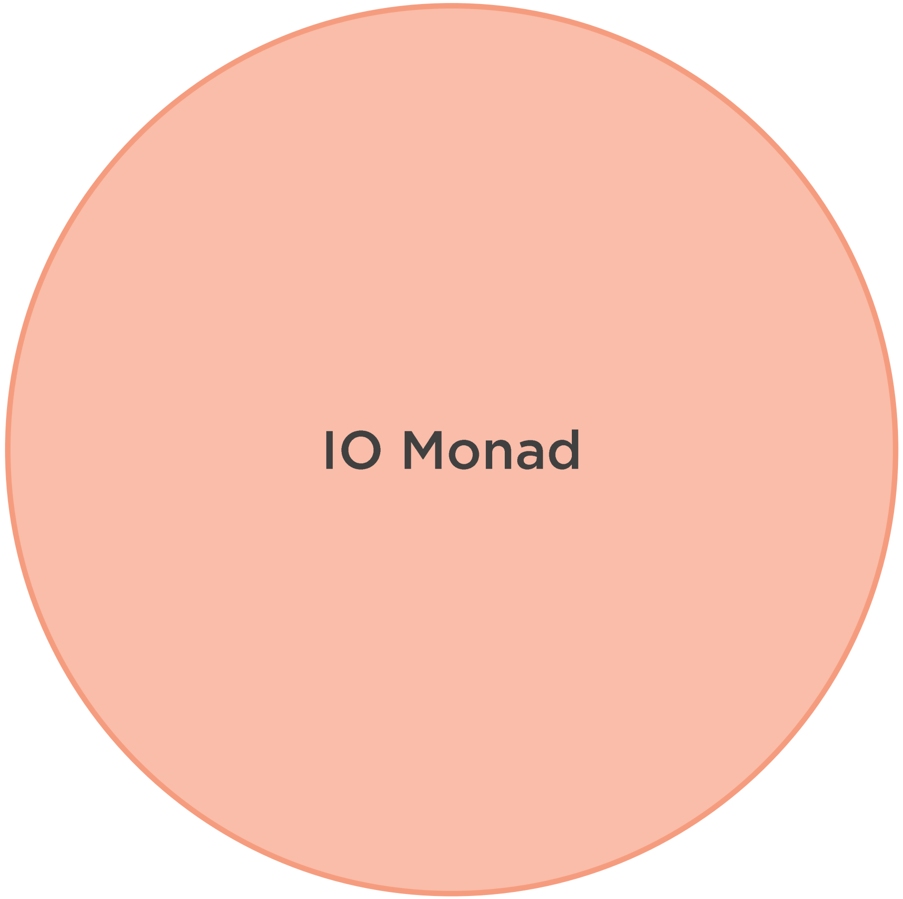
```
Effect program = Effect
                    .of( () -> read() )
                    .flatMap( x -> write(x) );

program.run();
```

Referential Transparency

Monads for composition

Purity via laziness



IO Monad

Implementing an IO Monad

Void

```
void writeFile(String content, String file) {
    // ...
}
```

Void

```
void writeFile(void) { // Compiler error
   // ...
}
```

Void Type

```
Callable<Void> callable = new Callable<Void>() {
   @Override
   public Void call() {
      System.out.println("Returning a Void type");
      return null;
   }
};
```

Course Summary

Course Summary

A pure function

- Has a single responsibility
- Has no side effects
- Is referentially transparent

Course Summary

Functional programming techniques

- Immutability
- High-order functions
- Currying
- Recursion
- Lazy evaluation

Course Summary

Composition

- Nesting functions, passing the result of one as the input of the next

High-order functions

- Take a function as their input
- Return a function as their output
- Do both

Course Summary

Functions with several arguments

- Functions of tuples
- Functions returning functions

Currying

- One-argument functions

Partial application

- Supplying fewer arguments

Argument order

- From the most specific to the least specific

Course Summary

Replace loops with recursion

- It can cause a StackOverflowException

Tail-recursive functions

- The recursive call is the last line of the function
- Use an accumulator to carry intermediate state
- Implement TCO with thunks and trampolines

Course Summary

Nulls make code dishonest

- Use Optional to explicitly indicate the absence of a value

For handling errors

- Prefer total functions that return a type that groups a valid value or an error

Course Summary

Monads are computational contexts

- They are a parameterized type
- They have a unit function to put a value inside
- They have a bind function to apply a function to transform the value

Course Summary

Laziness

- About indicating that we may do something sometime in the future

To handle effects functionally

- Implement a lazy function that when executed will produce an effect
- Wrap it in a monad to compose many effects while keeping them isolated

Where to Go from Here

Thinking in Terms of Pure Functions

Practice

Start small

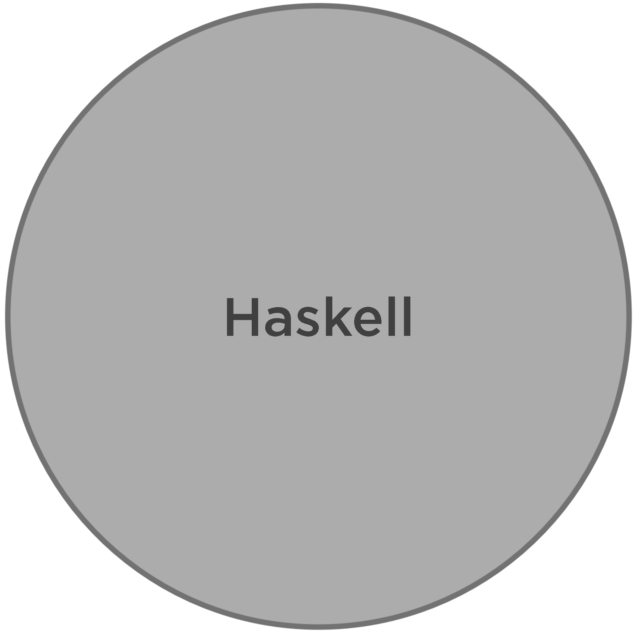
Functional-friendly Languages in the JVM

Scala

Kotlin

Clojure

Outside the JVM



Haskell

Recommended Third-party Library

Vavr

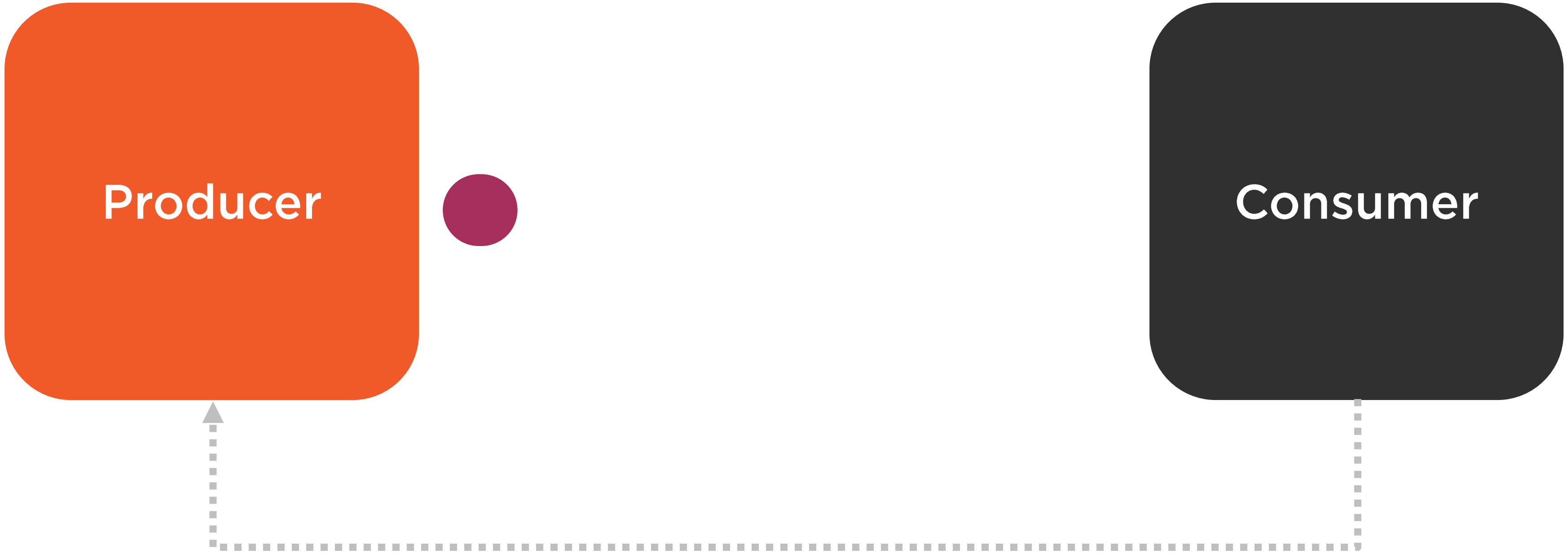
- Immutable data structures
- Tuples
- Functional interfaces
- More types

Reactive Programming Frameworks

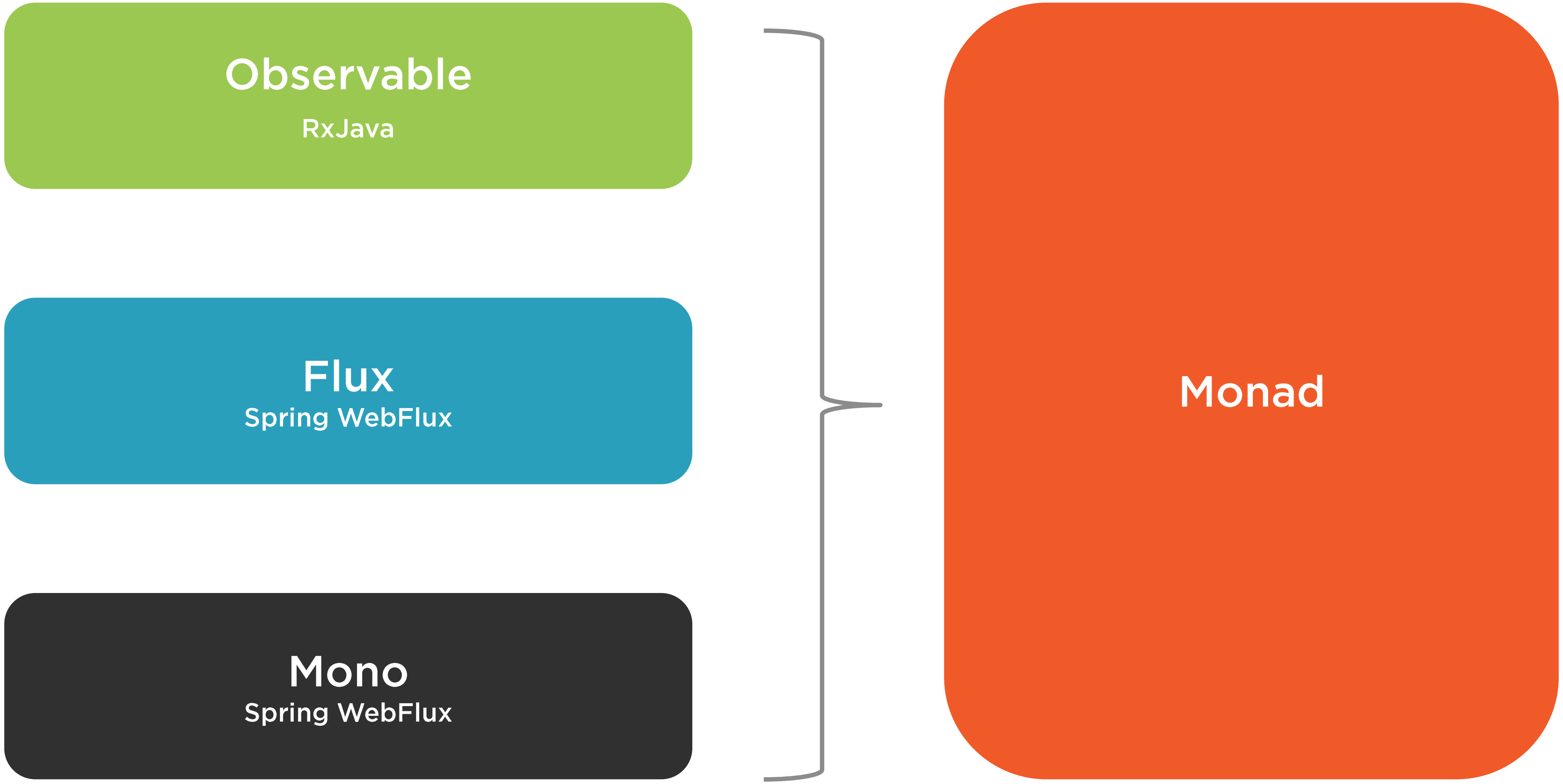
RxJava

Spring WebFlux

Reactive Programming



Request when ready
request(n)



Observable

RxJava

Flux

Spring WebFlux

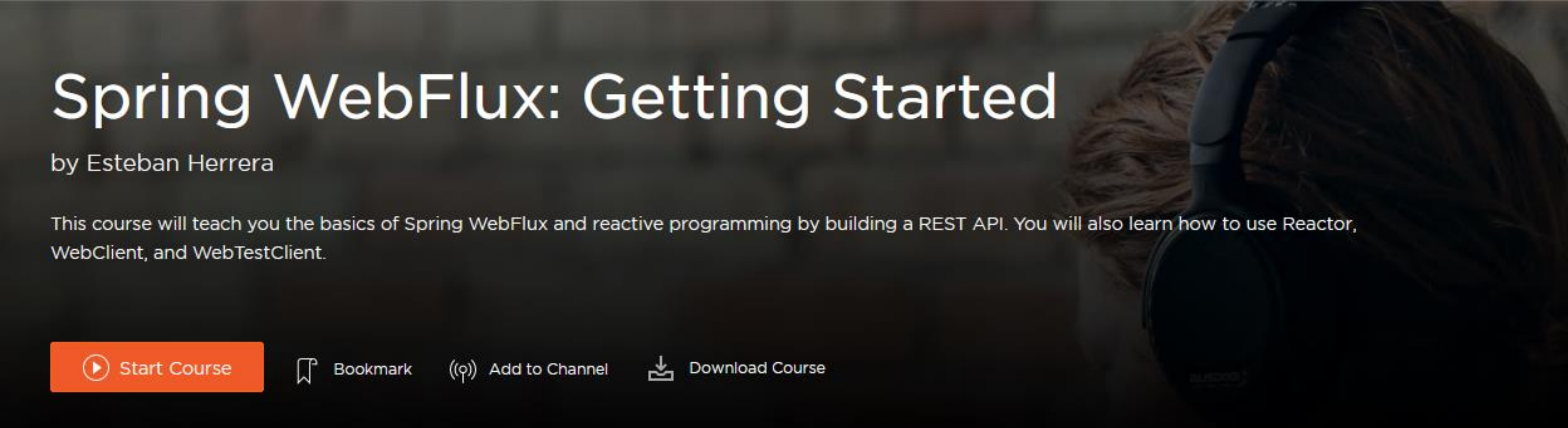
Mono

Spring WebFlux

Monad

Reactive Programming

Reactive Functional Programming



Spring WebFlux: Getting Started

by Esteban Herrera

This course will teach you the basics of Spring WebFlux and reactive programming by building a REST API. You will also learn how to use Reactor, WebClient, and WebTestClient.

http://bit.ly/webflux-gs

Thank you