Reactive Spring Boot In Action

Özgür Rahmi Dönmez

Sercan Çelenk

Contents

- Imperative vs Functional Programming vs Reactive Programming
- Features of Reactive Systems
- Handling Events with Callbacks & Functional Composition
- Monads
- Monad : Map, Flatmap, Unit functions
- Reactive Streams as Monad
- Hot & Cold Streams - Subscription
- Demo

Programming Paradigms

- Imperative Programming (Java,C...)
- Functional Programming (Haskell,Scala...)
- Logic Programming (Prolog)

Ortogonal :

- Object Oriented Programming

Imperative Programming

- It is about changing the state of a program using series of instructions (statements) :
  i. Assignments
  ii. Control Flow Statements (if then else,loops,break,continue,return)

- Many things have to happen in a certain order (sequential)

- At each stage the programmer must keep in mind the current state and how it will be modified

Imperative Programming

- We need other techniques for defining high level abstractions & compositions to build new complicated programs from existing ones!

- We need mathematical proof that our code will work

Imperative Programming

Imperative Programming is based on mutation !

Mutation : “Changing the value while keeping the identity is the same”

Mathematical Theories consists of :

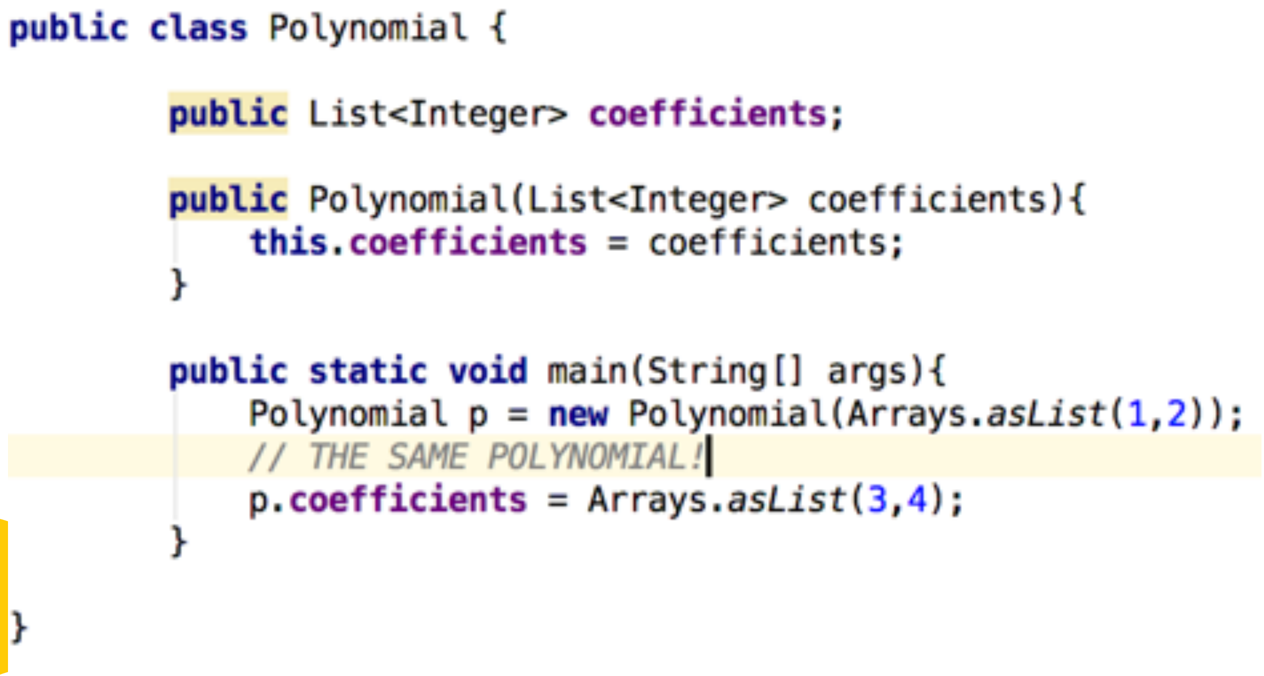
I. One or more data types
II. Operation on these types
III. Laws describes relationship between values and operations

***** Does not describe mutations!***

Imperative Programming

Polynomials : (a*x + b)

It does not define a operator to change coefficients while keeping the polynomial the same!!



```java
public class Polynomial {

        public List<Integer> coefficients;

        public Polynomial(List<Integer> coefficients){
            this.coefficients = coefficients;
        }

        public static void main(String[] args){
            Polynomial p = new Polynomial(Arrays.asList(1,2));
            // THE SAME POLYNOMIAL!
            p.coefficients = Arrays.asList(3,4);
        }

}
```

Functional Programming

FP means :

- Programming without mutable variables, assignments, loops & other control structures
- Writing your program in terms of functions

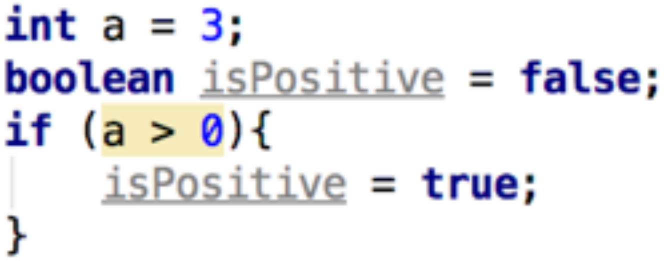
Functional Programming

Pure Functions :

- Does not rely on external state
- Has no side effects
- Evaluation order is not important
- Favours Parallelism
- No locking issues with objects that never change!
- You can pass immutable objects to functions safely, you know it wont be mutated
- Fits to big data model (map , reduce)

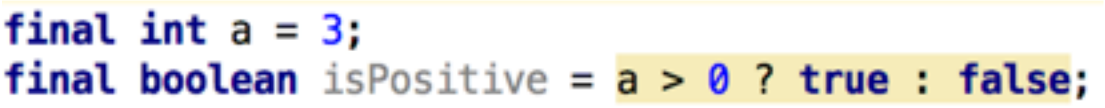
No Ifs

with if



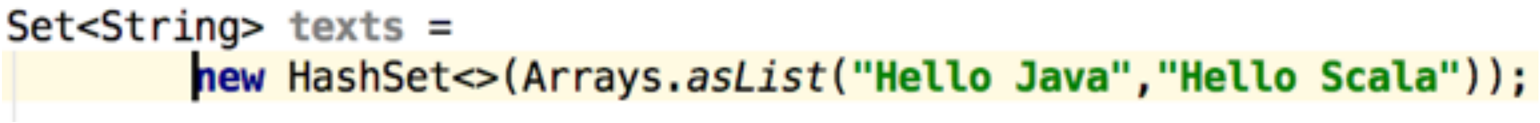
```java
int a = 3;
boolean isPositive = false;
if (a > 0){
    isPositive = true;
}
```

without if



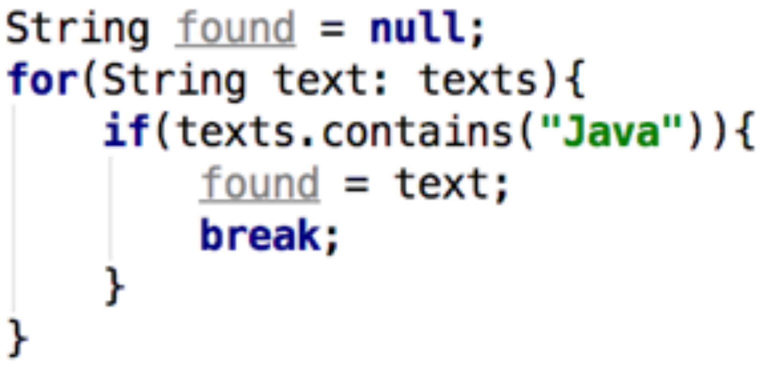
```java
final int a = 3;
final boolean isPositive = a > 0 ? true : false;
```

No loops



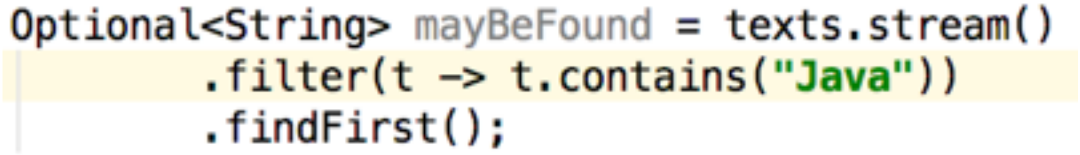
```
Set<String> texts =
        new HashSet<>(Arrays.asList("Hello Java","Hello Scala"));
```

with loop



```
String found = null;
for(String text: texts){
    if(texts.contains("Java")){
        found = text;
        break;
    }
}
```

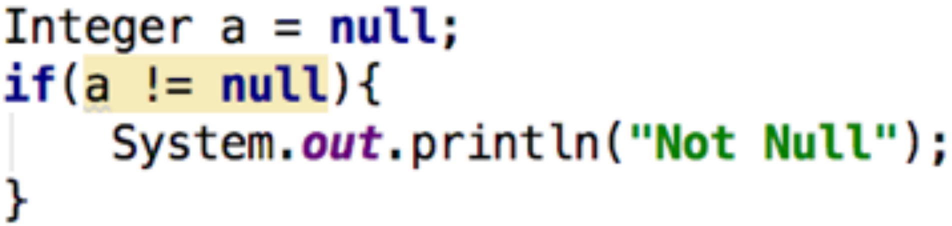
without loop



```
Optional<String> mayBeFound = texts.stream()
        .filter(t -> t.contains("Java"))
        .findFirst();
```

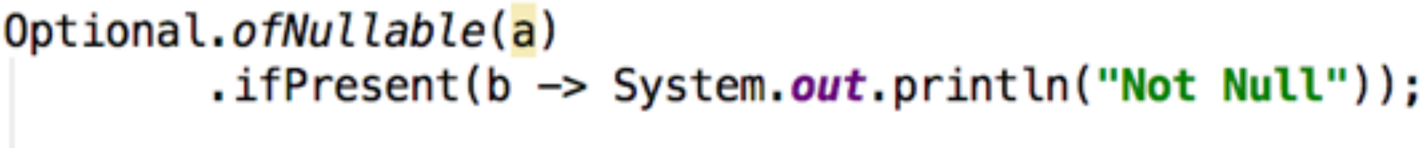
No null checks

with null check



```
Integer a = null;
if(a != null){
    System.out.println("Not Null");
}
```

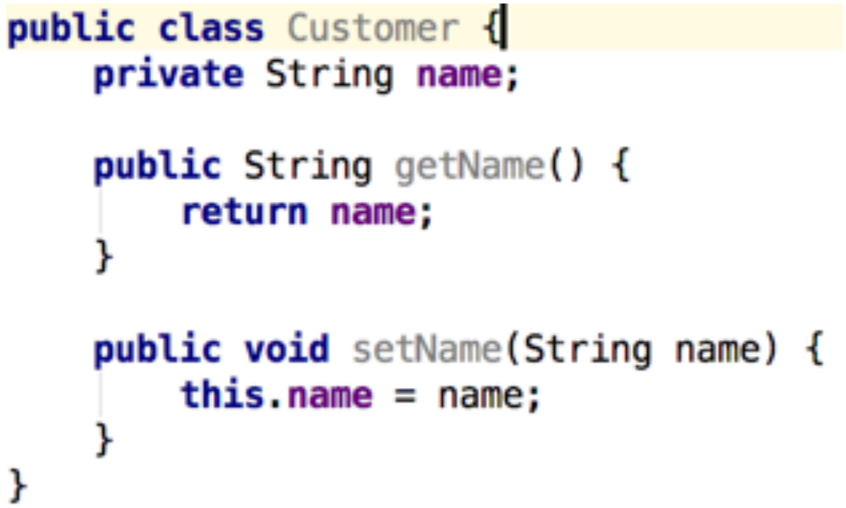
without null check



```
Optional.ofNullable(a)
        .ifPresent(b -> System.out.println("Not Null"));
```

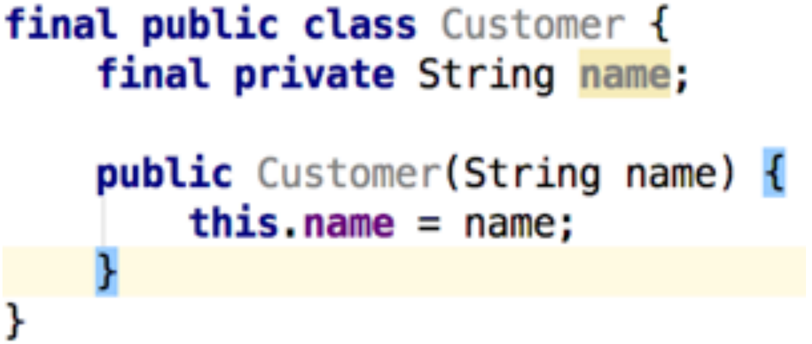
No mutable objects

Mutable



```java
public class Customer {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

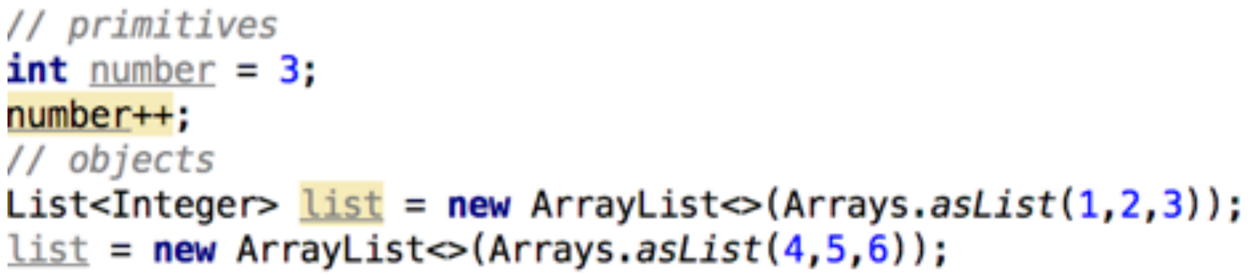
Immutable



```java
final public class Customer {
    final private String name;

    public Customer(String name) {
        this.name = name;
    }
}
```

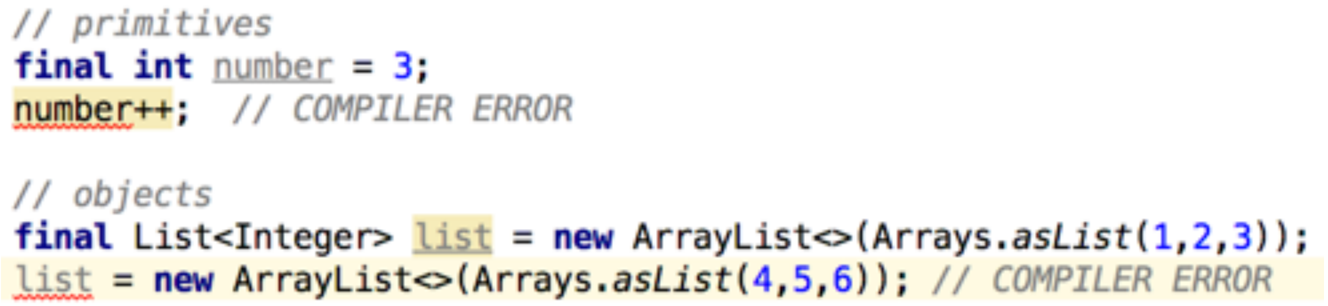
Use final keyword

Without final



```
// primitives
int number = 3;
number++;
// objects
List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3));
list = new ArrayList<>(Arrays.asList(4,5,6));
```

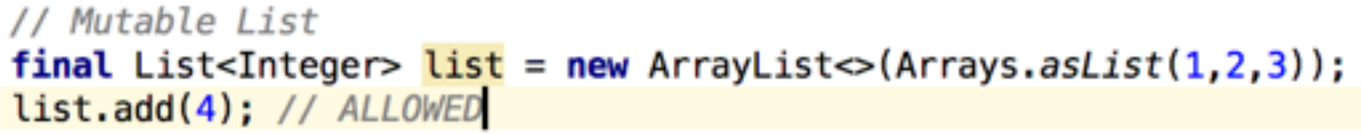
With final



```
// primitives
final int number = 3;
number++;  // COMPILER ERROR

// objects
final List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3));
list = new ArrayList<>(Arrays.asList(4,5,6)); // COMPILER ERROR
```

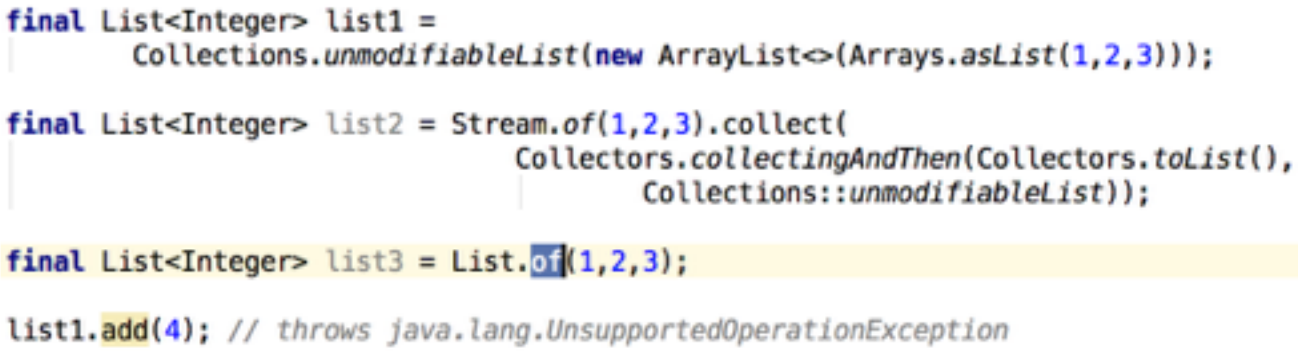
No mutable collections

With mutable collections



```
// Mutable List
final List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3));
list.add(4); // ALLOWED
```

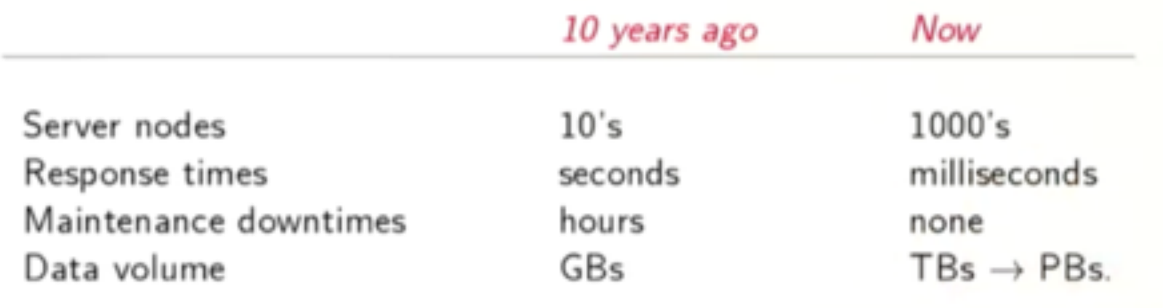
Without mutable collections



```
final List<Integer> list1 =
        Collections.unmodifiableList(new ArrayList<>(Arrays.asList(1,2,3)));

final List<Integer> list2 = Stream.of(1,2,3).collect(
                                Collectors.collectingAndThen(Collectors.toList(),
                                        Collections::unmodifiableList));

final List<Integer> list3 = List.of(1,2,3);

list1.add(4); // throws java.lang.UnsupportedOperationException
```

Why Reactive Programming?

- Changing Requirements



	10 years ago	*Now*
Server nodes	10's	1000's
Response times	seconds	milliseconds
Maintenance downtimes	hours	none
Data volume	GBs	TBs → PBs.

- We need new architecture : Reactive Applications

Reactive Systems

- Reactive : Readily responsive to stimulus

I. React to messages (Message-driven)

Relies on : Async message passing between components that ensures loose coupling, isolation and location transparency

II. React to load (Elastic)

“Stays responsive under varying load”

III. React to failures (Resilient)

“Stays responsive in the face of failure”

IV. React to users in a timely manner (Responsive)

Why Reactive Programming?

- **Classical model :** Systems composed of multiple threads, which communicate with shared , synchronised state

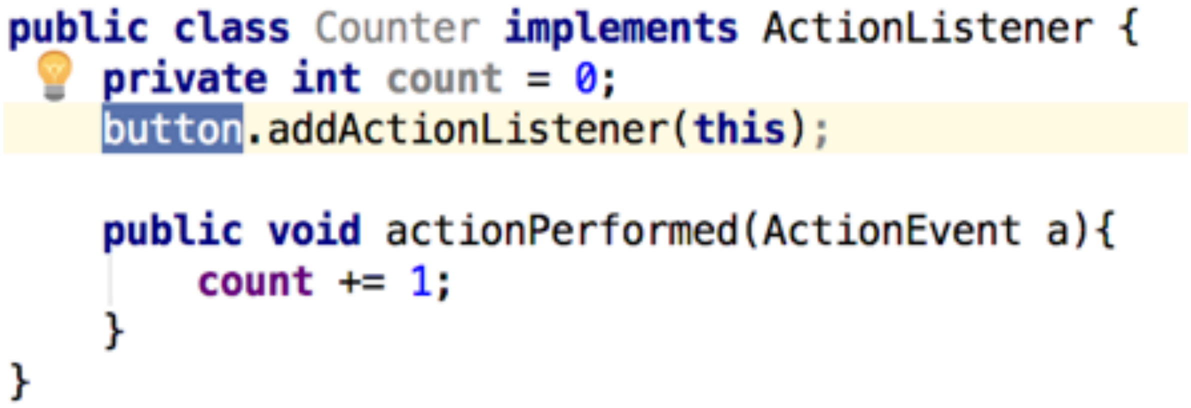
☑ Thread per connection

▷ Strong coupling, hard to scale

- **Reactive model :** Composed of loosely coupled event handlers

▷ Events handled async without blocking

Event Driven : With Callbacks



```java
public class Counter implements ActionListener {
    private int count = 0;
    button.addActionListener(this);

    public void actionPerformed(ActionEvent a){
        count += 1;
    }
}
```

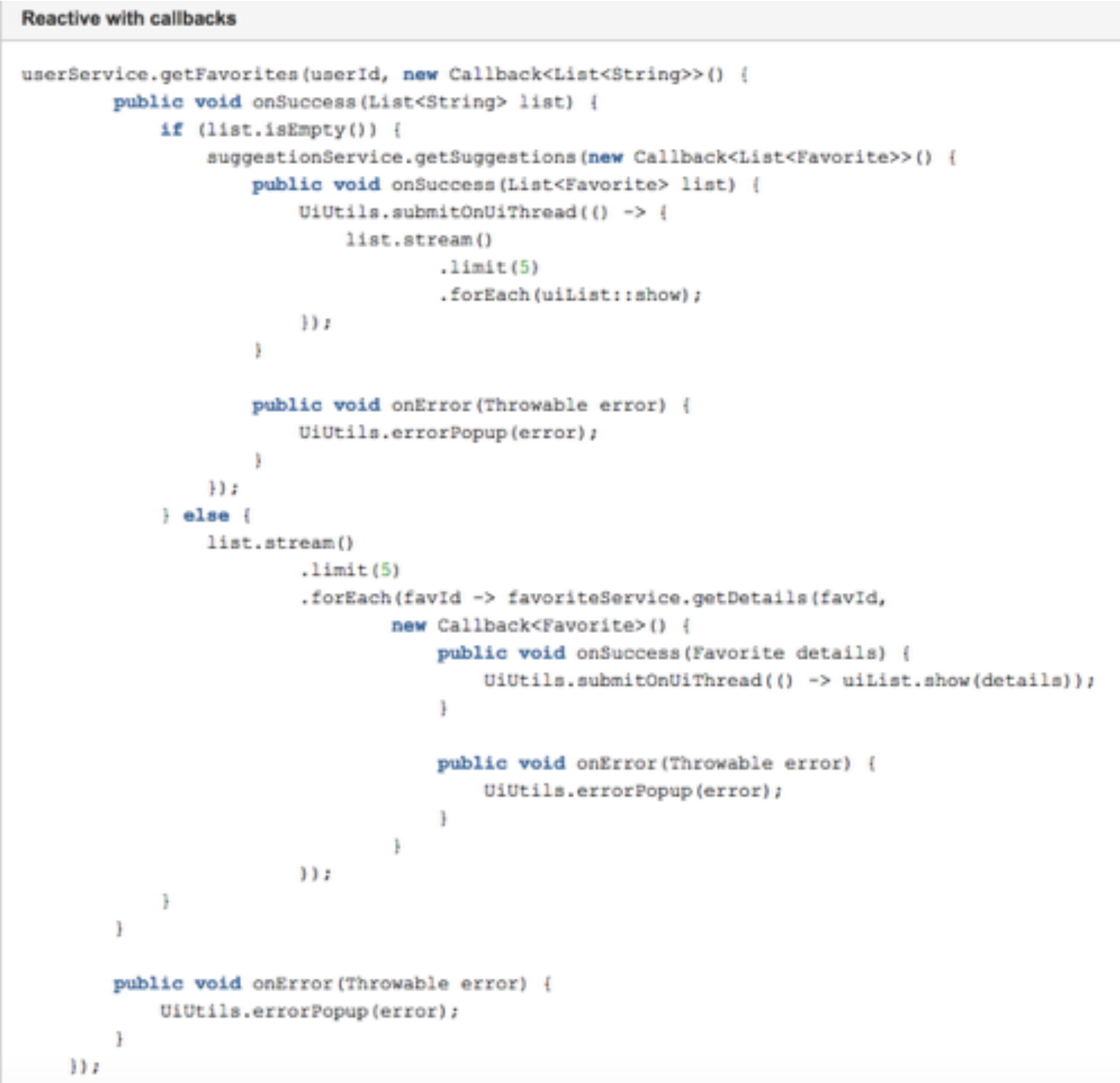
Observer Pattern!

Problems :

- Needs shared mutable state
- No higher abstractions/composition with listeners
- Callback hell!

Event Driven : With Callbacks

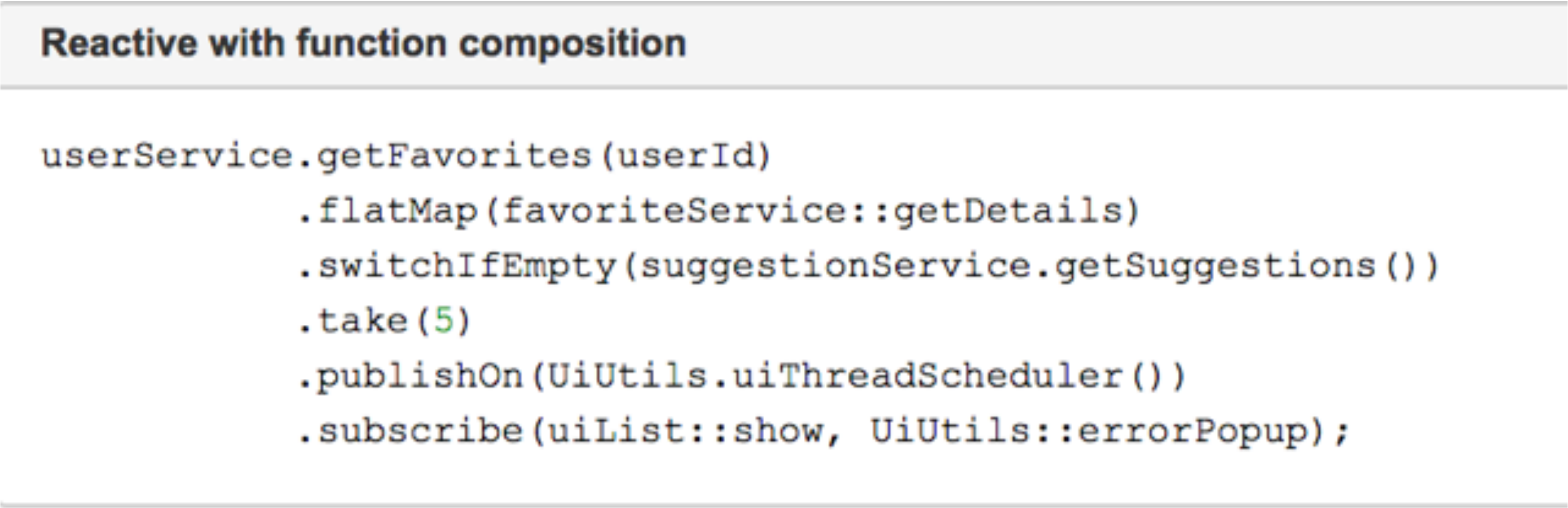
TURKCELL



Reactive with callbacks

```
userService.getFavorites(userId, new Callback<List<String>>() {
        public void onSuccess(List<String> list) {
            if (list.isEmpty()) {
                suggestionService.getSuggestions(new Callback<List<Favorite>>() {
                    public void onSuccess(List<Favorite> list) {
                        UiUtils.submitOnUiThread(() -> {
                            list.stream()
                                    .limit(5)
                                    .forEach(uiList::show);
                        });
                    }

                    public void onError(Throwable error) {
                        UiUtils.errorPopup(error);
                    }
                });
            } else {
                list.stream()
                        .limit(5)
                        .forEach(favId -> favoriteService.getDetails(favId,
                                new Callback<Favorite>() {
                                    public void onSuccess(Favorite details) {
                                        UiUtils.submitOnUiThread(() -> uiList.show(details));
                                    }

                                    public void onError(Throwable error) {
                                        UiUtils.errorPopup(error);
                                    }
                                }
                        ));
            }
        }

        public void onError(Throwable error) {
            UiUtils.errorPopup(error);
        }
    });
```

Event Driven : With Functional Composition



Reactive with function composition

```
userService.getFavorites(userId)
              .flatMap(favoriteService::getDetails)
              .switchIfEmpty(suggestionService.getSuggestions())
              .take(5)
              .publishOn(UiUtils.uiThreadScheduler())
              .subscribe(uiList::show, UiUtils::errorPopup);
```

Composibility : Ability to orchestrate multiple asynchronous tasks using results from previous tasks

Monads - map,flatMap,unit functions

Monad

- Kind of container for values
- Wrapper
- Allows us to make transformations on values without always getting the value inside
- It is not always necessary ask for the value from the "container" every time you want to do something with them.

  A. the value inside may not exist

  B. May not have been evaluated yet

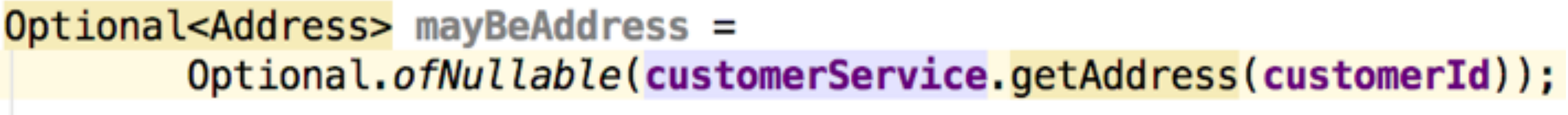
Monads - map,flatMap,unit functions

Monad constraints :

1. **Unit function :** A way to put a value into the monad : Wrapping part, constructor that takes single value.
2. **Map function :** A way to perform an operation on the values and return a new monad with the new values.
3. **Flatten :** A way to flatten nested monads. This is called "flatten"

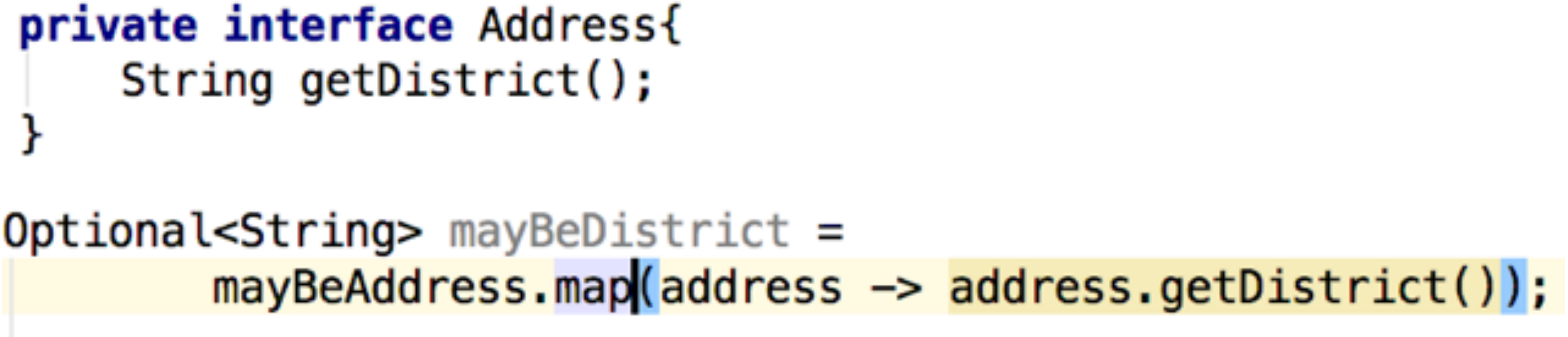
Monads - map,flatMap,unit functions

Unit function for Optional



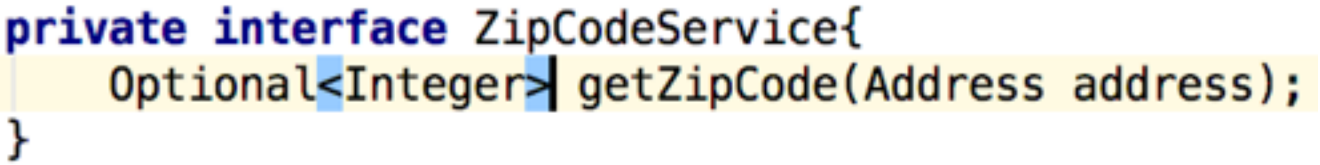
```
Optional<Address> mayBeAddress =
        Optional.ofNullable(customerService.getAddress(customerId));
```

Map function for Optional

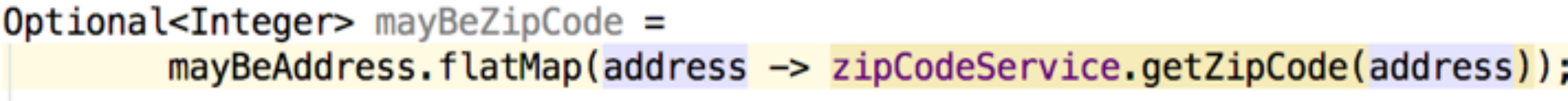


```
private interface Address{
    String getDistrict();
}

Optional<String> mayBeDistrict =
        mayBeAddress.map(address -> address.getDistrict());
```

Flatmap function for Optional



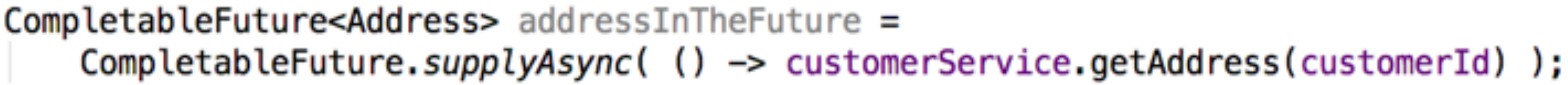
```
private interface ZipCodeService{
    Optional<Integer> getZipCode(Address address);
}
```



```
Optional<Integer> mayBeZipCode =
        mayBeAddress.flatMap(address -> zipCodeService.getZipCode(address));
```

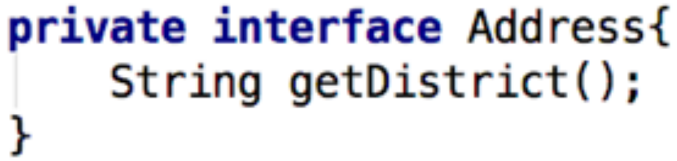
Monads - map,flatMap,unit functions

Unit function for Future

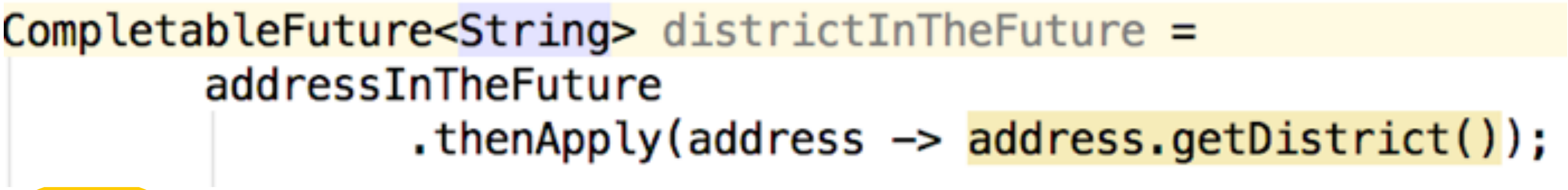


```
CompletableFuture<Address> addressInTheFuture =
    CompletableFuture.supplyAsync( () -> customerService.getAddress(customerId) );
```

Map function for Future

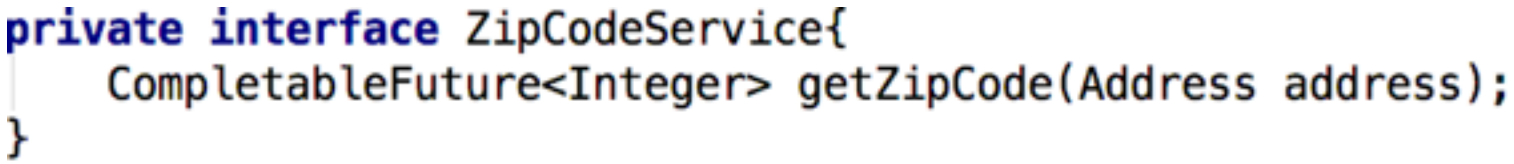


```
private interface Address{
    String getDistrict();
}
```

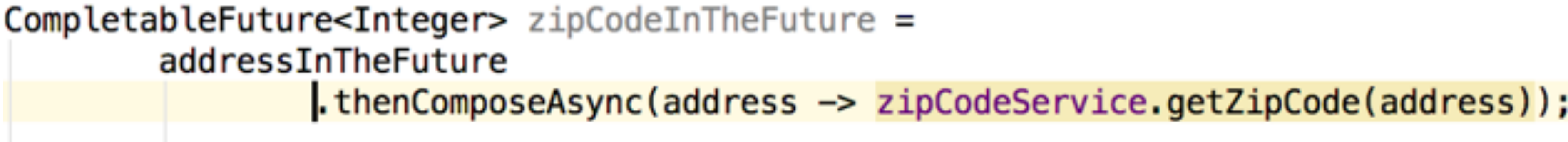


```
CompletableFuture<String> districtInTheFuture =
        addressInTheFuture
                .thenApply(address -> address.getDistrict());
```

Flatmap function for Future



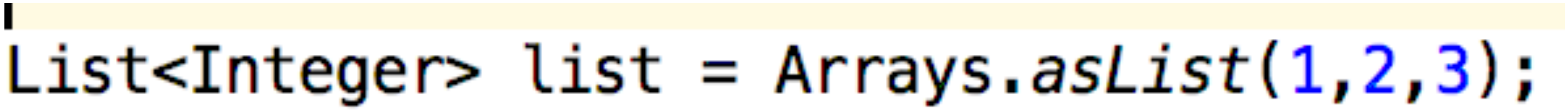
```
private interface ZipCodeService{
    CompletableFuture<Integer> getZipCode(Address address);
}
```



```
CompletableFuture<Integer> zipCodeInTheFuture =
        addressInTheFuture
                .thenComposeAsync(address -> zipCodeService.getZipCode(address));
```

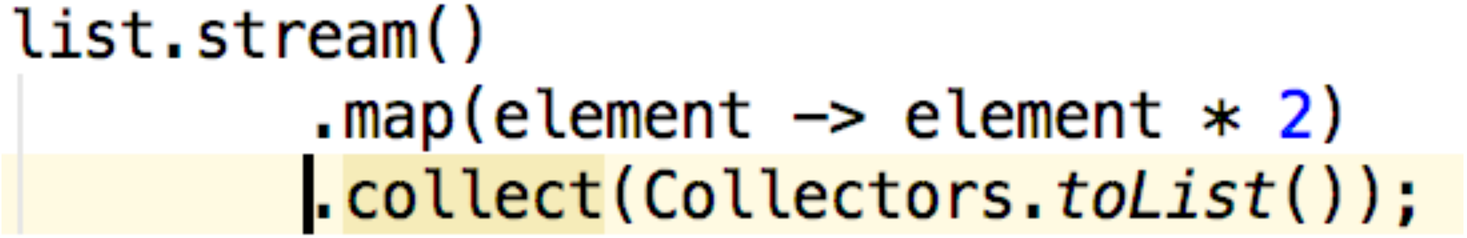
Monads - map,flatMap,unit functions

Unit function for List (as stream)



```java
List<Integer> list = Arrays.asList(1,2,3);
```

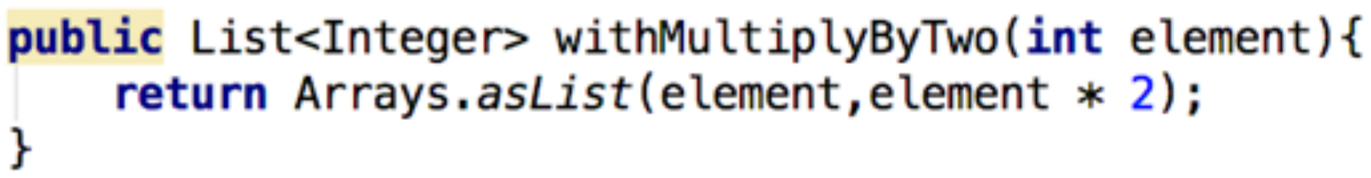
Map function for List (as stream)



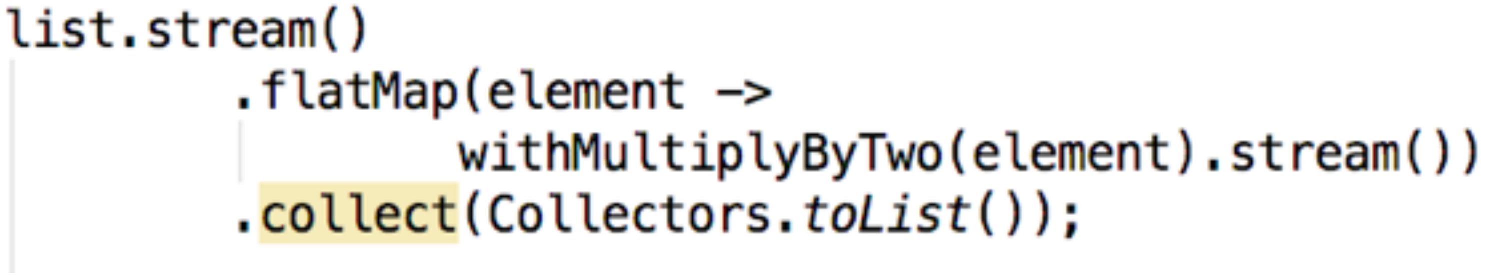
```java
list.stream()
        .map(element -> element * 2)
        .collect(Collectors.toList());
```

Monads - map,flatMap,unit functions

Flatmap function for List (as stream)

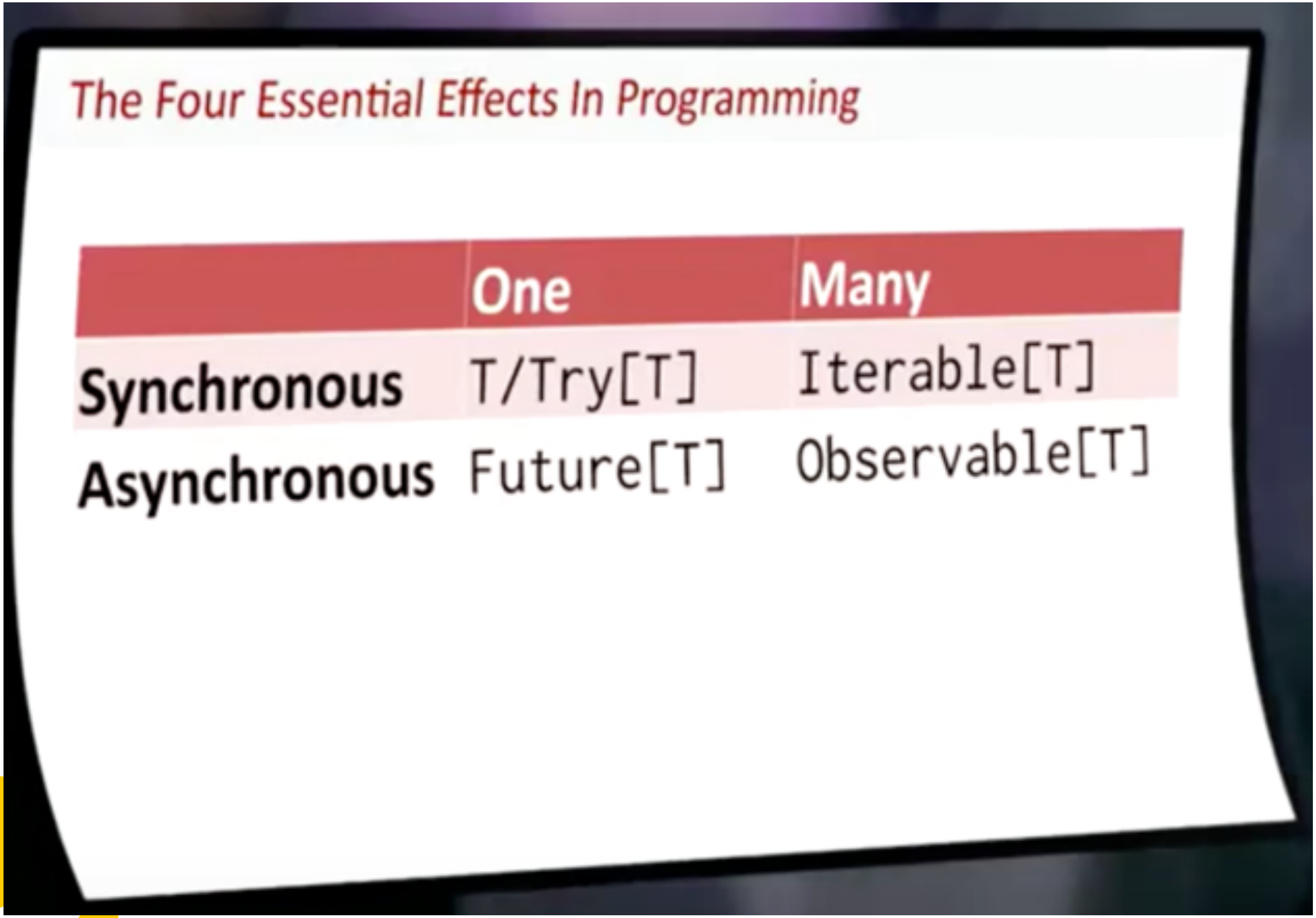


```java
public List<Integer> withMultiplyByTwo(int element){
    return Arrays.asList(element,element * 2);
}
```



```java
list.stream()
        .flatMap(element ->
                withMultiplyByTwo(element).stream())
        .collect(Collectors.toList());
```

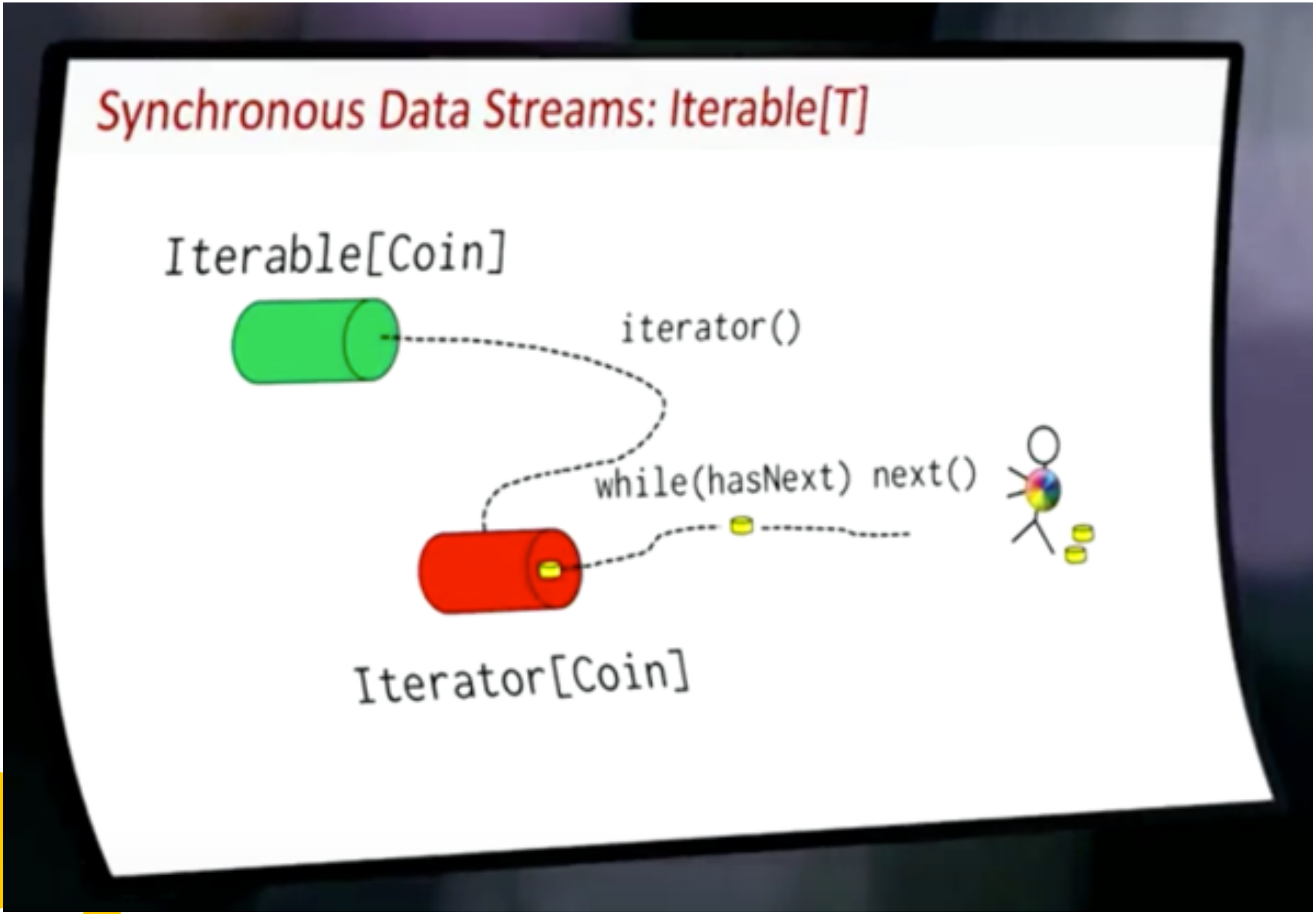
What is Stream (Observable | Flux)



The Four Essential Effects In Programming

	One	Many
Synchronous	T/Try[T]	Iterable[T]
Asynchronous	Future[T]	Observable[T]

What is Stream (Observable | Flux)



Synchronous Data Streams: Iterable[T]

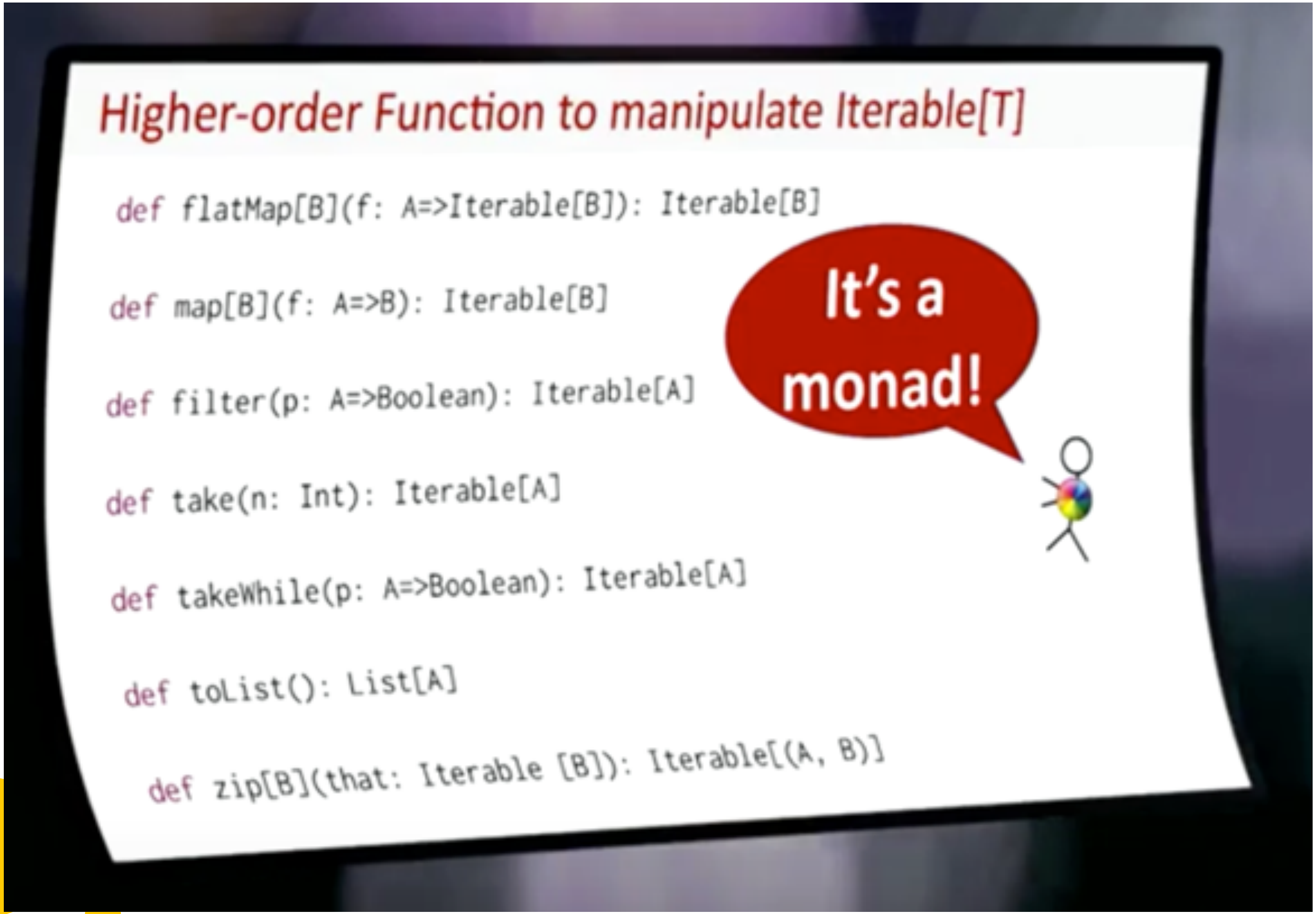
What is Stream (Observable | Flux)

- Iterators -> Synch, pull based (on demand)

we block when next()

- Reactive Streams -> Async , push based

What is Stream (Observable | Flux)

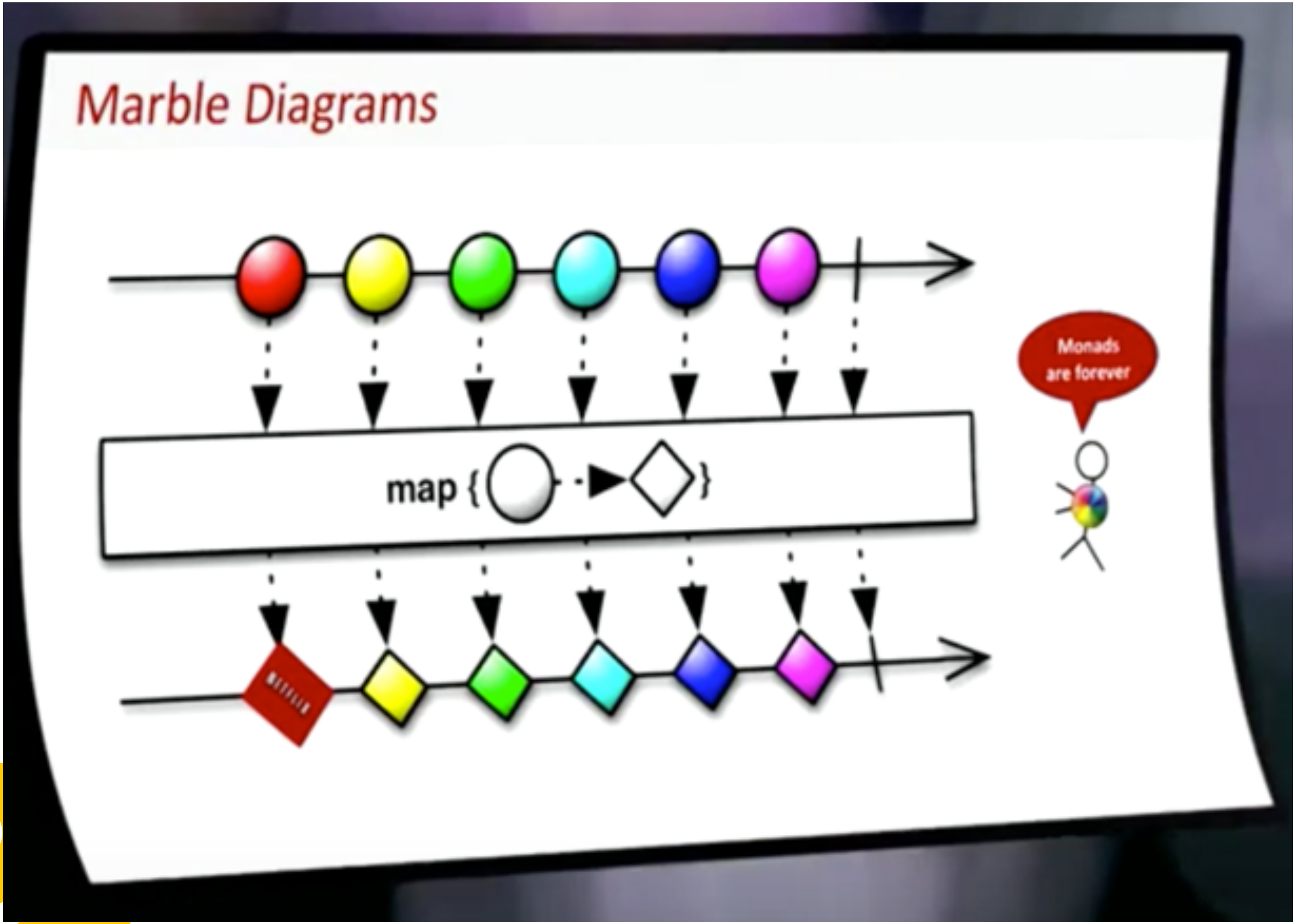


Higher-order Function to manipulate Iterable[T]

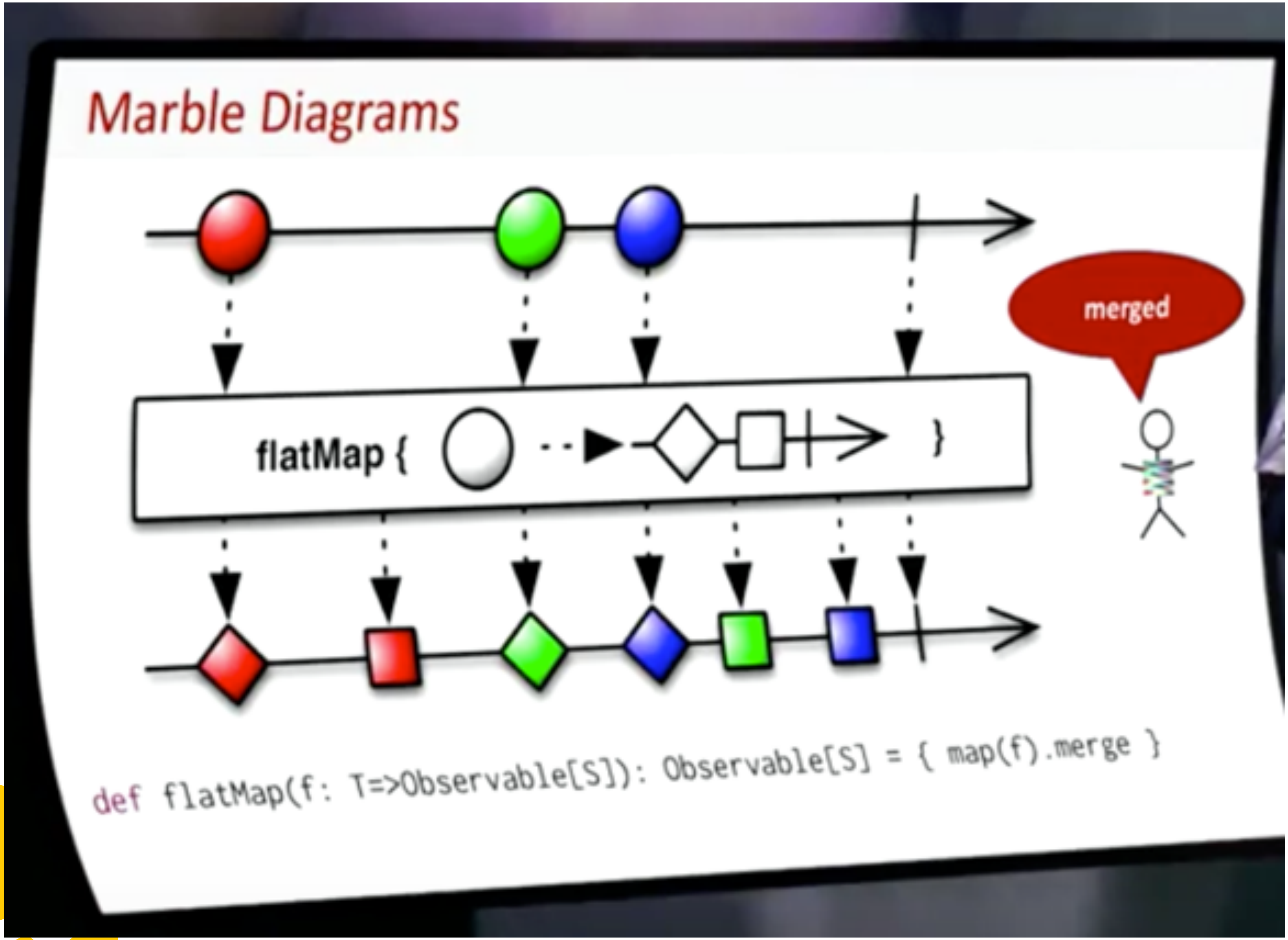
```
def flatMap[B](f: A=>Iterable[B]): Iterable[B]

def map[B](f: A=>B): Iterable[B]

def filter(p: A=>Boolean): Iterable[A]

def take(n: Int): Iterable[A]

def takeWhile(p: A=>Boolean): Iterable[A]

def toList(): List[A]

def zip[B](that: Iterable [B]): Iterable[(A, B)]
```

It's a monad!

What is Stream (Observable | Flux)



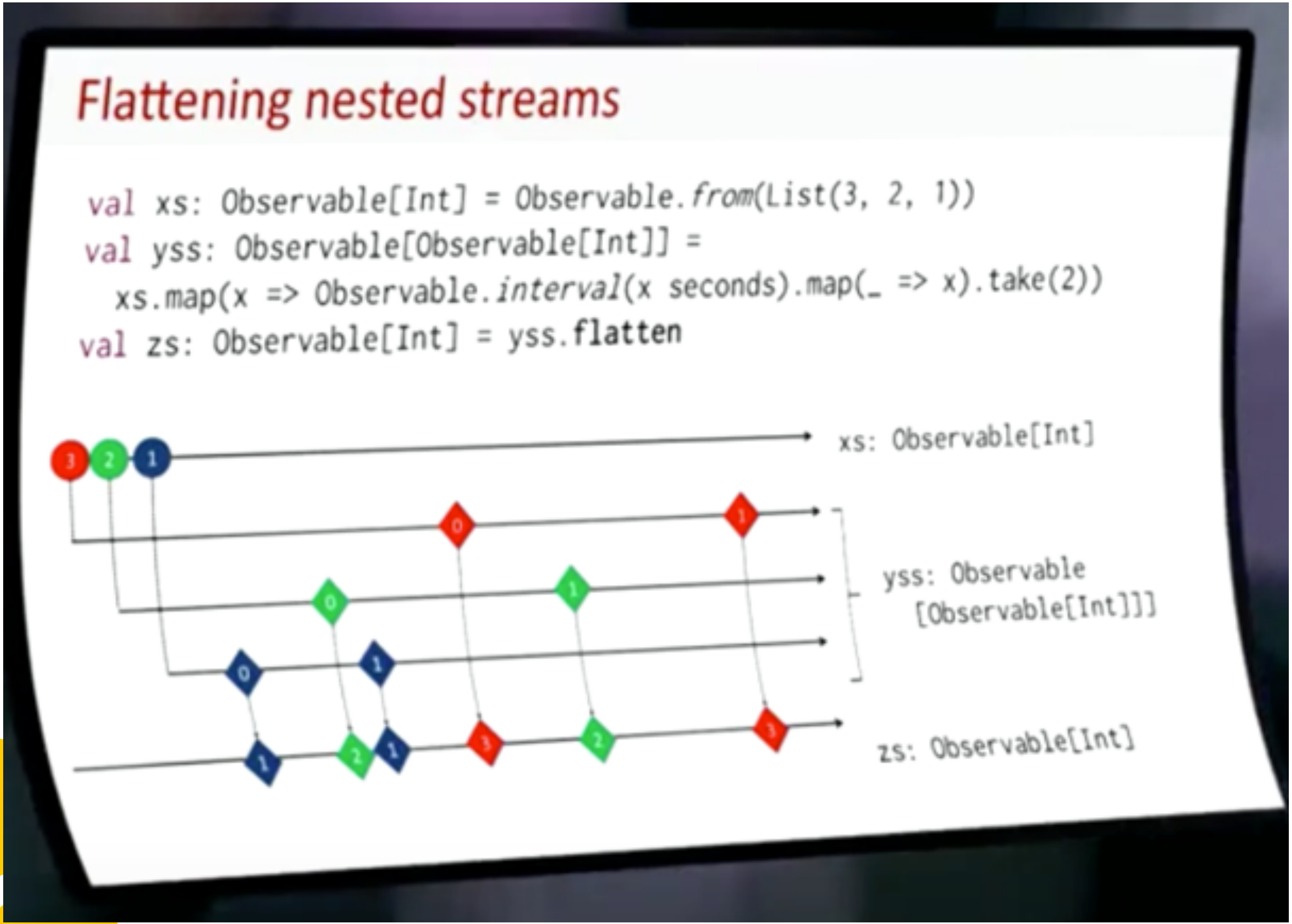
What is Stream (Observable | Flux)



Marble Diagrams

```
def flatMap(f: T=>Observable[S]): Observable[S] = { map(f).merge }
```

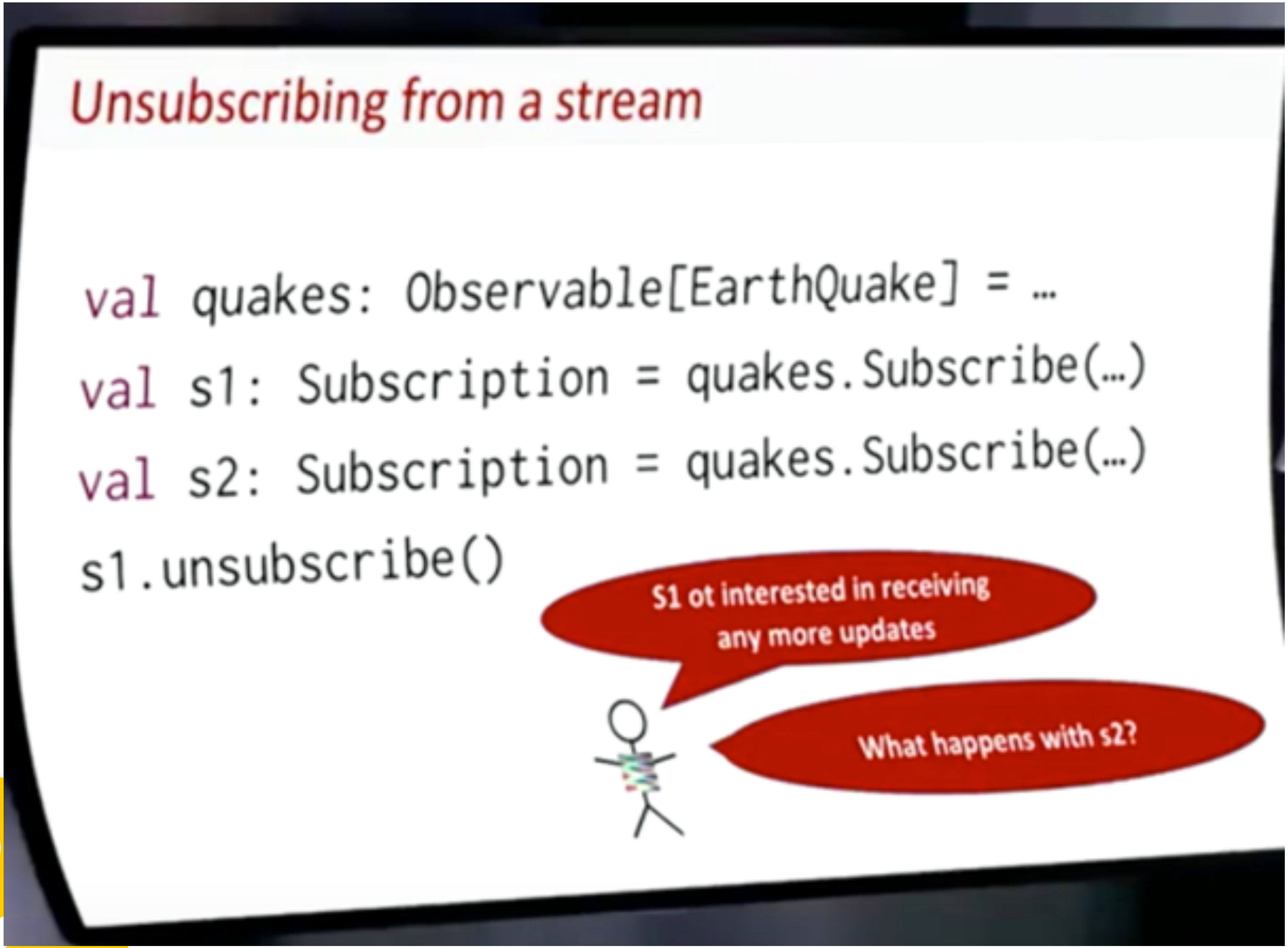
What is Stream (Observable | Flux)



Flattening nested streams

```
val xs: Observable[Int] = Observable.from(List(3, 2, 1))
val yss: Observable[Observable[Int]] =
  xs.map(x => Observable.interval(x seconds).map(_ => x).take(2))
val zs: Observable[Int] = yss.flatten
```

Subscription & Unsubscription



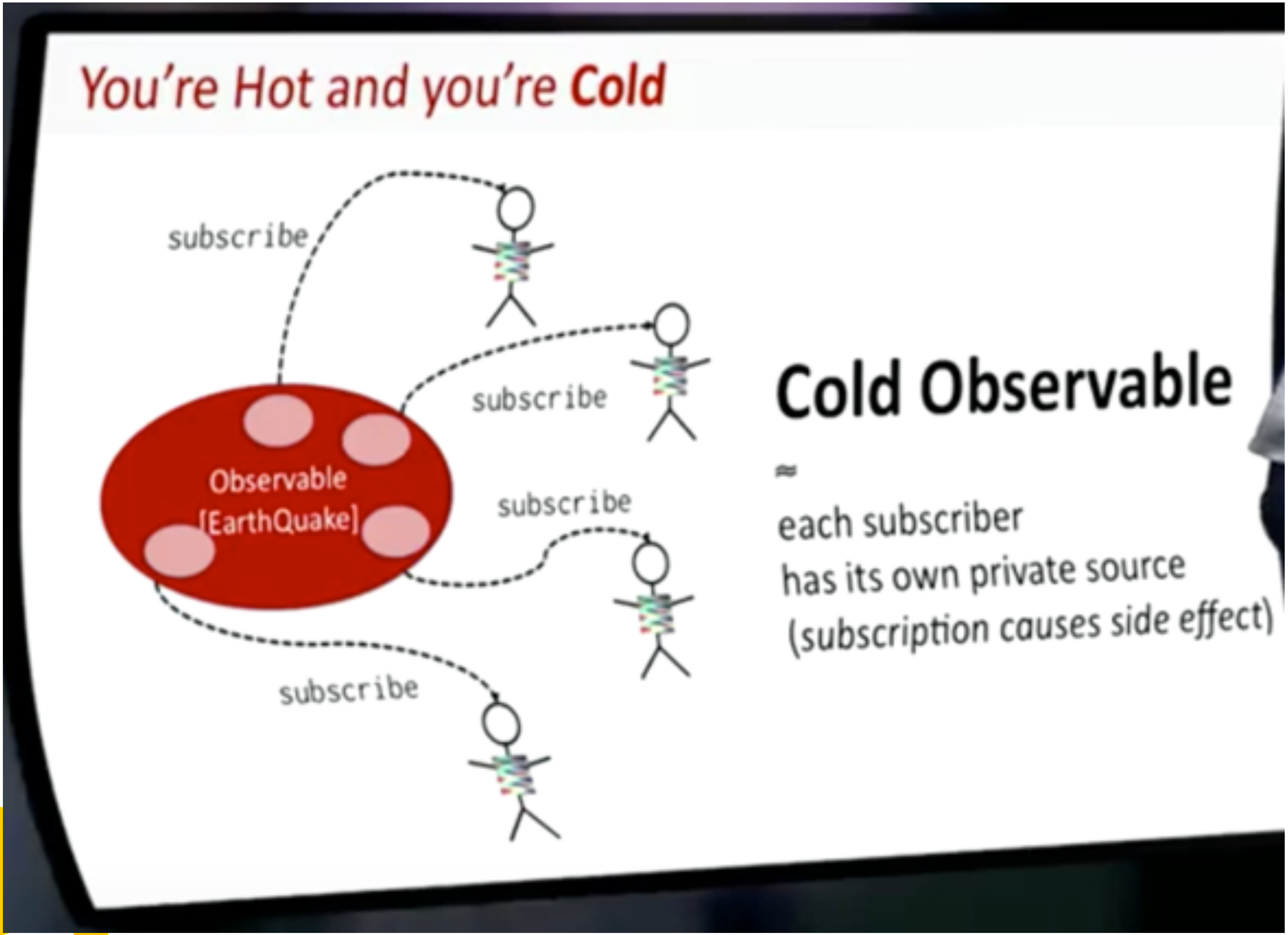
Unsubscribing from a stream

```
val quakes: Observable[EarthQuake] = …
val s1: Subscription = quakes.Subscribe(…)
val s2: Subscription = quakes.Subscribe(…)
s1.unsubscribe()
```

S1 ot interested in receiving any more updates

What happens with s2?

Subscription & Unsubscription



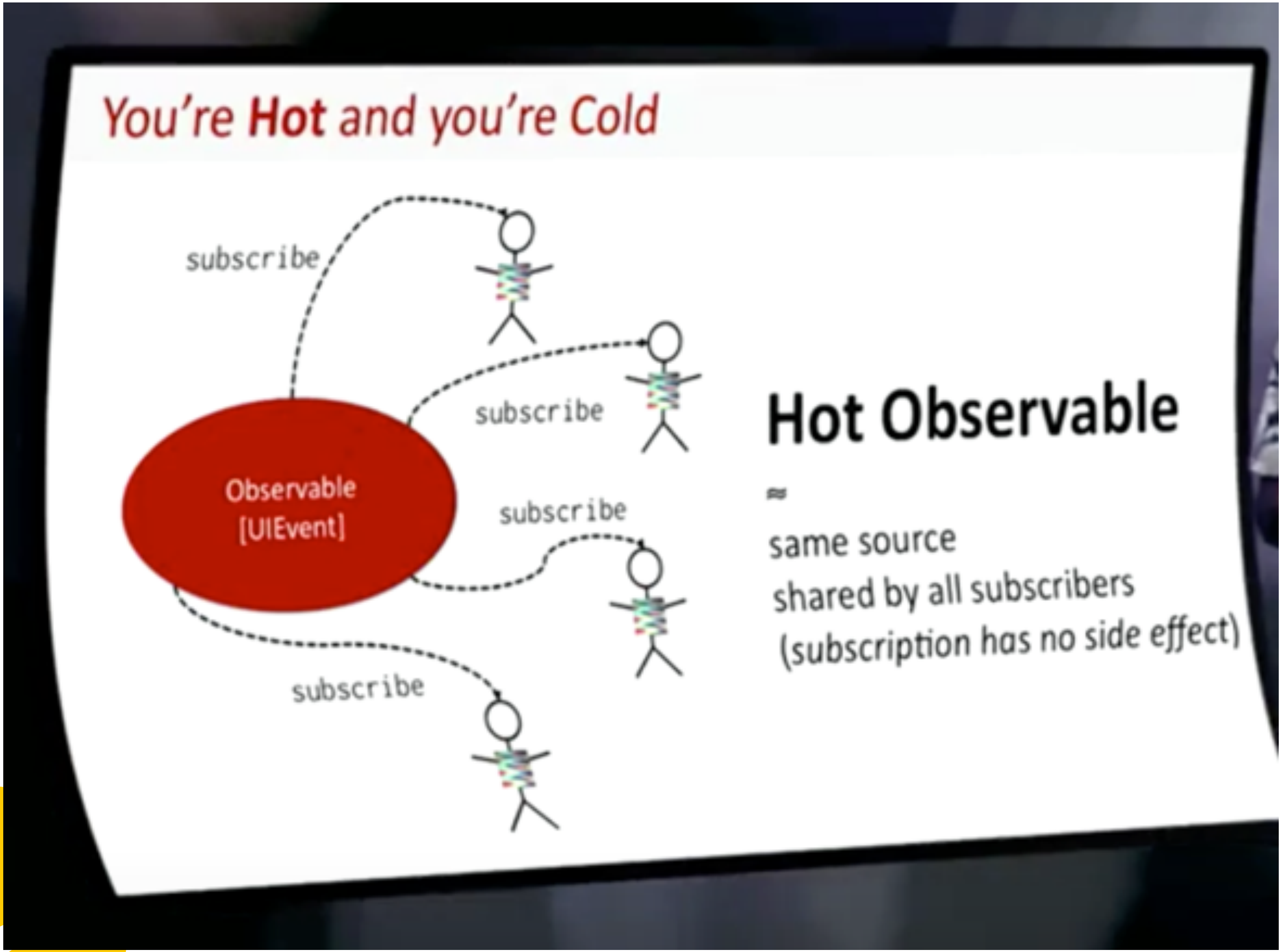
You're Hot and you're **Cold**

Cold Observable

≈

each subscriber
has its own private source
(subscription causes side effect)

Subscription & Unsubscription



You're **Hot** and you're Cold

subscribe
subscribe
subscribe
subscribe
Observable
[UIEvent]

Hot Observable

≈

same source

shared by all subscribers

(subscription has no side effect)

Backpressure

- What if Observable producer sends faster than we consume?

☑ We get Out of Memory

- So push-pull based streams is essential if our consumer is slower or faster!

☑ “Give me more items” or

☑ “Give me less items”

- Back pressure should also be non-blocking!