Beyond Java to Scala

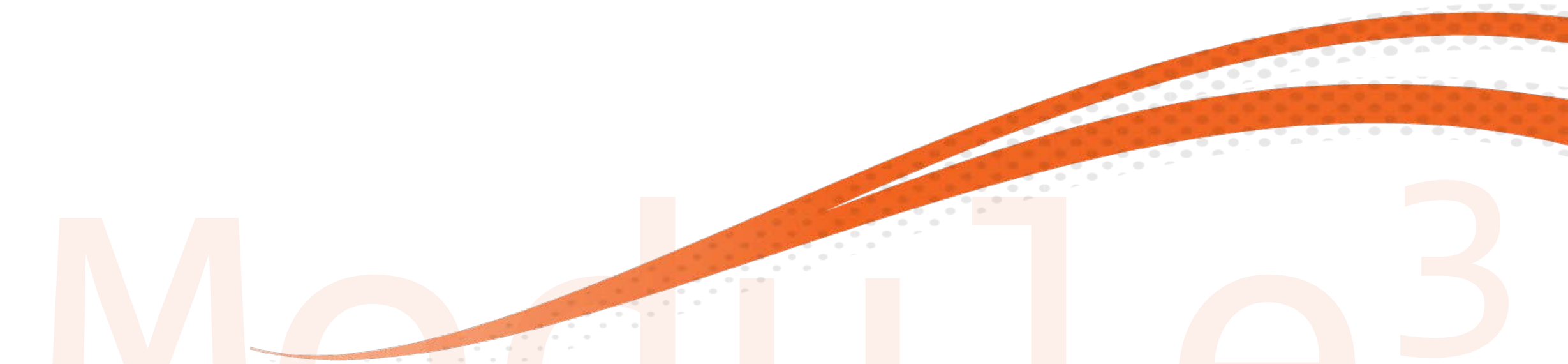
Toby Weston
http://baddotrobot.com
@jamanifin

pluralsight
hardcore dev and IT training

Beyond Java to Scala

Module Introduction

Let’s Get Started…



Beyond Java to Scala

Expressive Scala: Faking Function Calls

- The `apply` Method
- The `update` Method

Expressive Scala

- Faking Function Calls
- Faking Language Constructs
- Pattern Matching

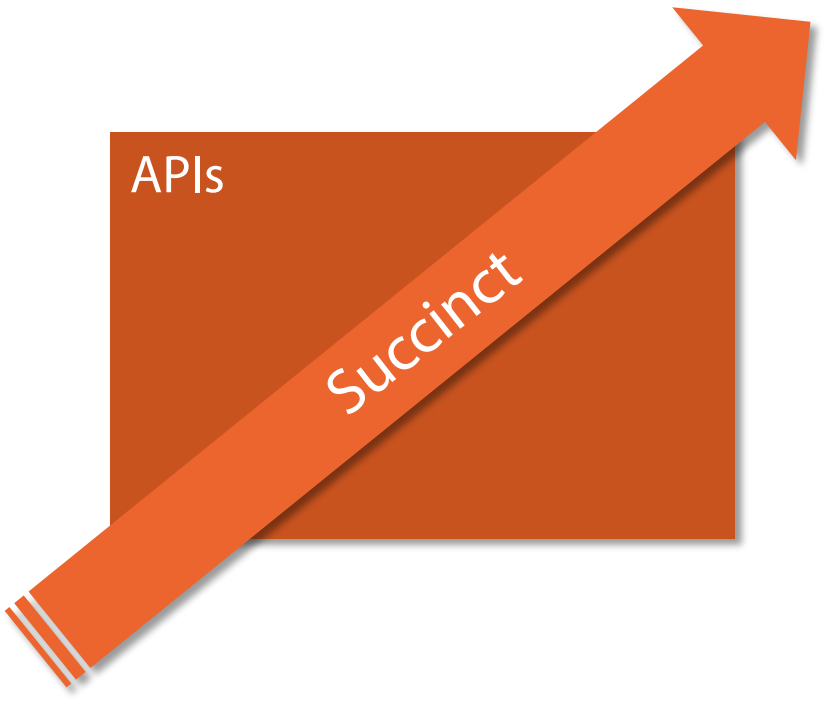
The “apply” method

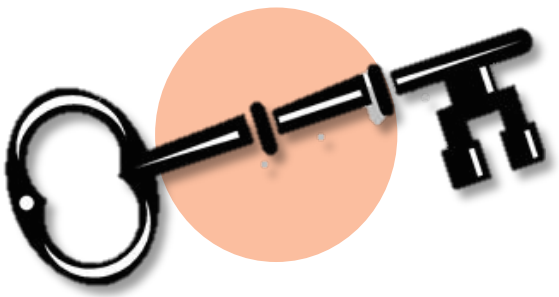
demos/01-customer.demo

DEMO

Multiple “apply” methods

Not just for factory methods


APIs
Succinct



Make any *method* call
look like a *function* call

```
Customer.apply("Rob Randal", "14 The Arches")
```

```
Customer("Rob Randal", "14 The Arches")
```

```
val add = new Adder()
add.apply(1, 3)
```

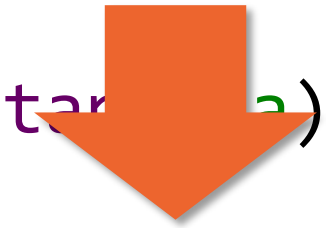
```
val add = new Adder()
add(1, 3)
```

```
array[access]
```

demos/02-arrays.demo

DEMO

The “update” method



```
instance.update(a, b)
```

_1

_1
_2

```
_1 = Map.Entry.getKey()
_2
```

```
_1 = Map.Entry.getKey()
_2 = Map.Entry.getValue()
```

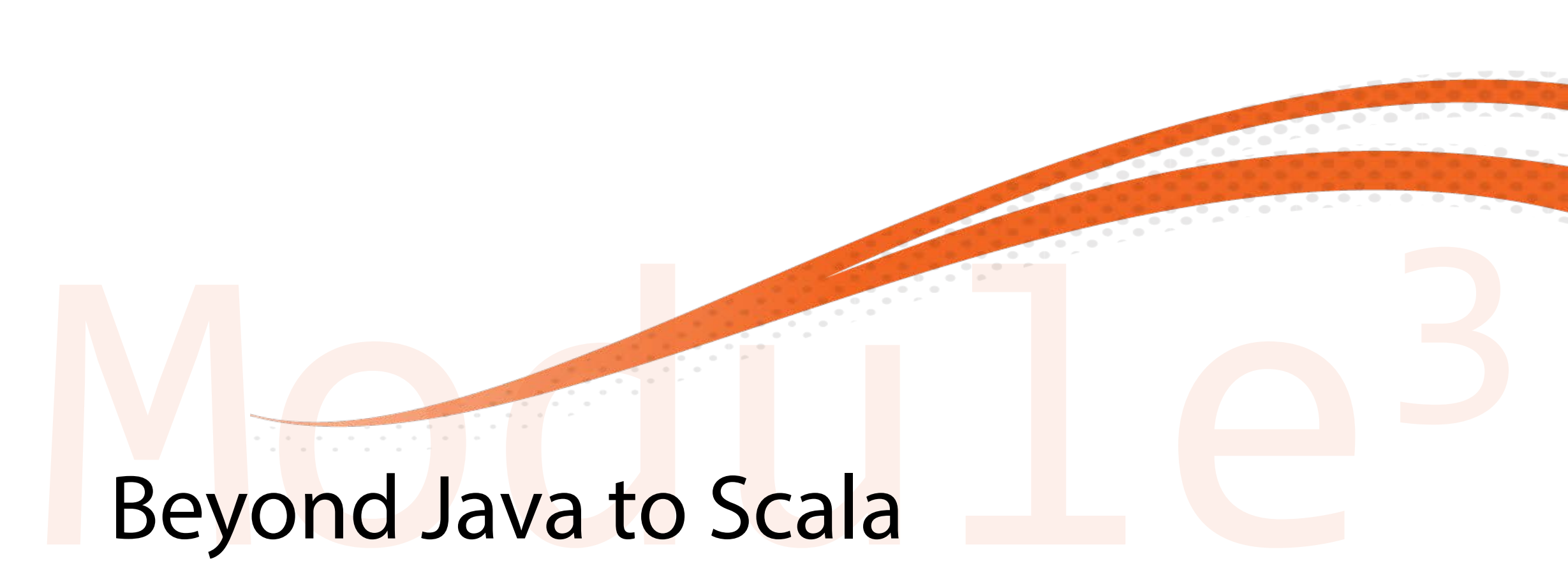
Summary

Expressive Scala

- Faking Function Calls
  - Not just for factory methods
  - Calling `apply` looks like a function call
  - Assignment operator & `update`

Expressive Scala

- Faking Function Calls
  - Not just for factory methods
  - Calling `apply` looks like a function call
  - Assignment operator & `update`
- Faking Language Constructs
- Pattern Matching



Beyond Java to Scala

Expressive Scala: Faking Language Constructs

- Curly Braces
- Higher-Order Functions
- Curried Functions

Expressive Scala

- Faking Function Calls
- Faking Language Constructs
- Pattern Matching

{}

( )

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly one argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround that argument instead of parentheses.

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround **that** argument instead of parentheses.

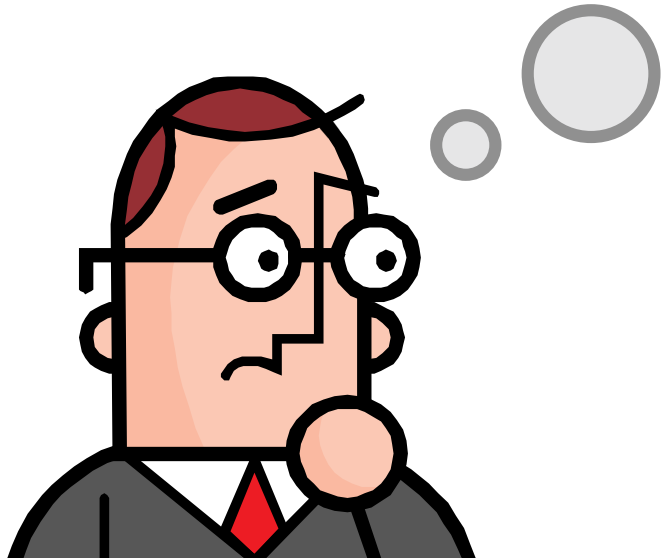
Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround **that** argument instead of parentheses.

Higher Order Functions

call-by-name

By-name Parameters = Lazy Parameters



By-name Parameters != Lazy

Expressive

Expressive &

Expressive & Natural

Control Structures
We're Already Used To

Curly Brace Rule –

In any method call, in which you pass in exactly **one** argument, you can use curly braces to surround **that** argument instead of parentheses.

currying

Currying –

The process of turning a function of two or more arguments into a series of functions, each taking a single argument.

$$f(a, b) = a + b$$

Original

$$f(a, b) = a + b$$

Curried

$$f(a)$$

Original

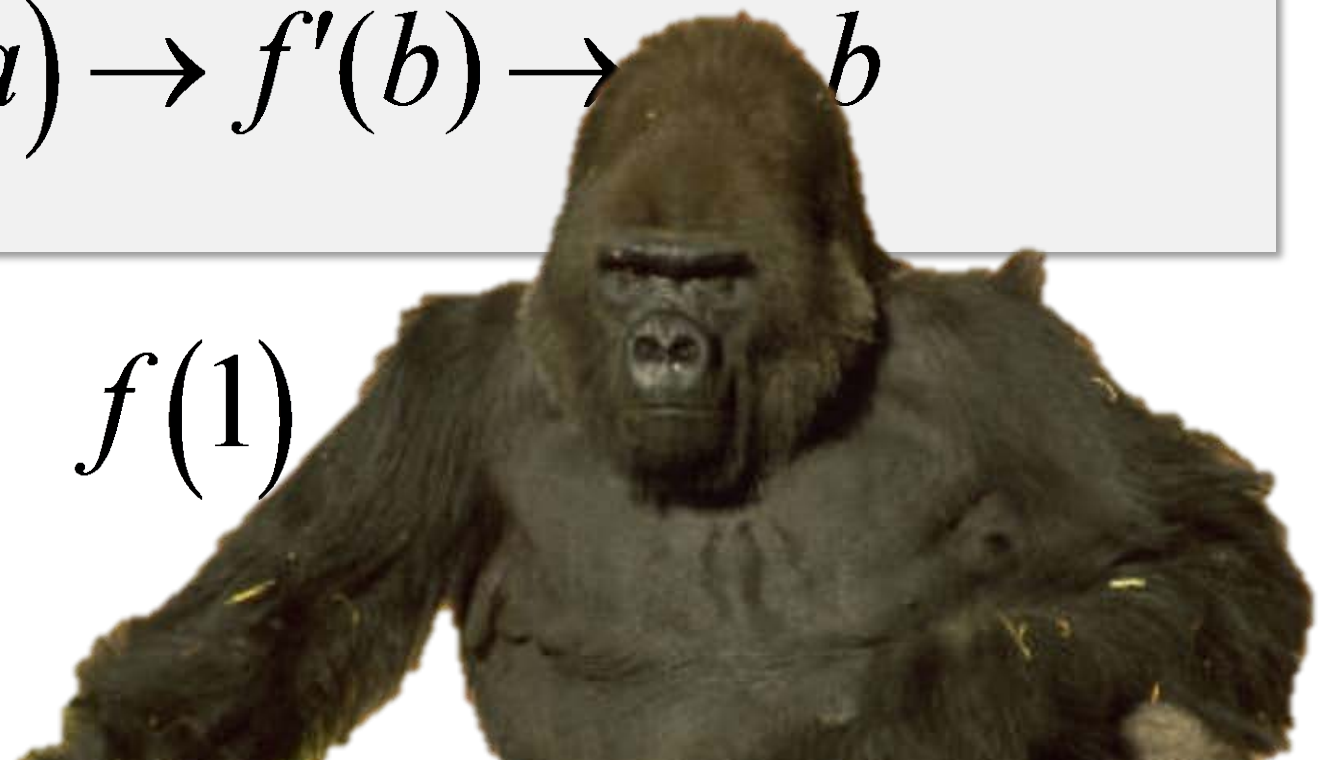
$$f(a, b) = a + b$$

Curried

$$f(a) \rightarrow f'(b) \rightarrow b$$

$$f(1)$$

Evaluation



Original

$$f(a, b) = a + b$$

Curried

$$f(a) = (b \mapsto (a + b))$$

Lambda Calculus

$$\lambda ab.\ a + b$$

Curried

$$\lambda a.\left(\lambda b.\left(a + b\right)\right)$$

Closure –

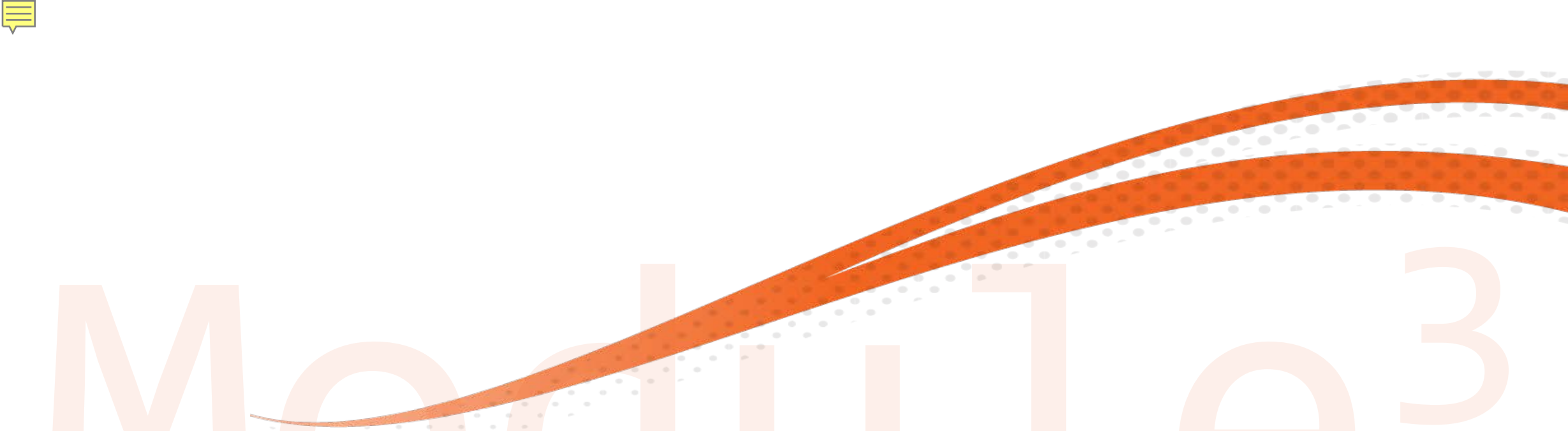
Capturing a value and making it available to a second function by "closing" over the environment.

See http://bit.ly/1AGOAnS

Summary

Expressive Scala

- Faking Function Calls
- Faking Language Constructs
  - Higher Order Functions
  - { The Curly Brace Rule }
  - Currying
- Pattern Matching



Beyond Java to Scala

Expressive Scala: Pattern Matching

- Switch-like Behaviour
- Anatomy of Patterns
- Literal, Constructor & Type Query
- Extractors

DEMO

- No fall through / no `break`
- A case match is an expression
- Flexible match conditions

- Guard conditions
- `MatchError`

- Compiler checked

What are “Patterns”?

```
value
```

```
value match
```

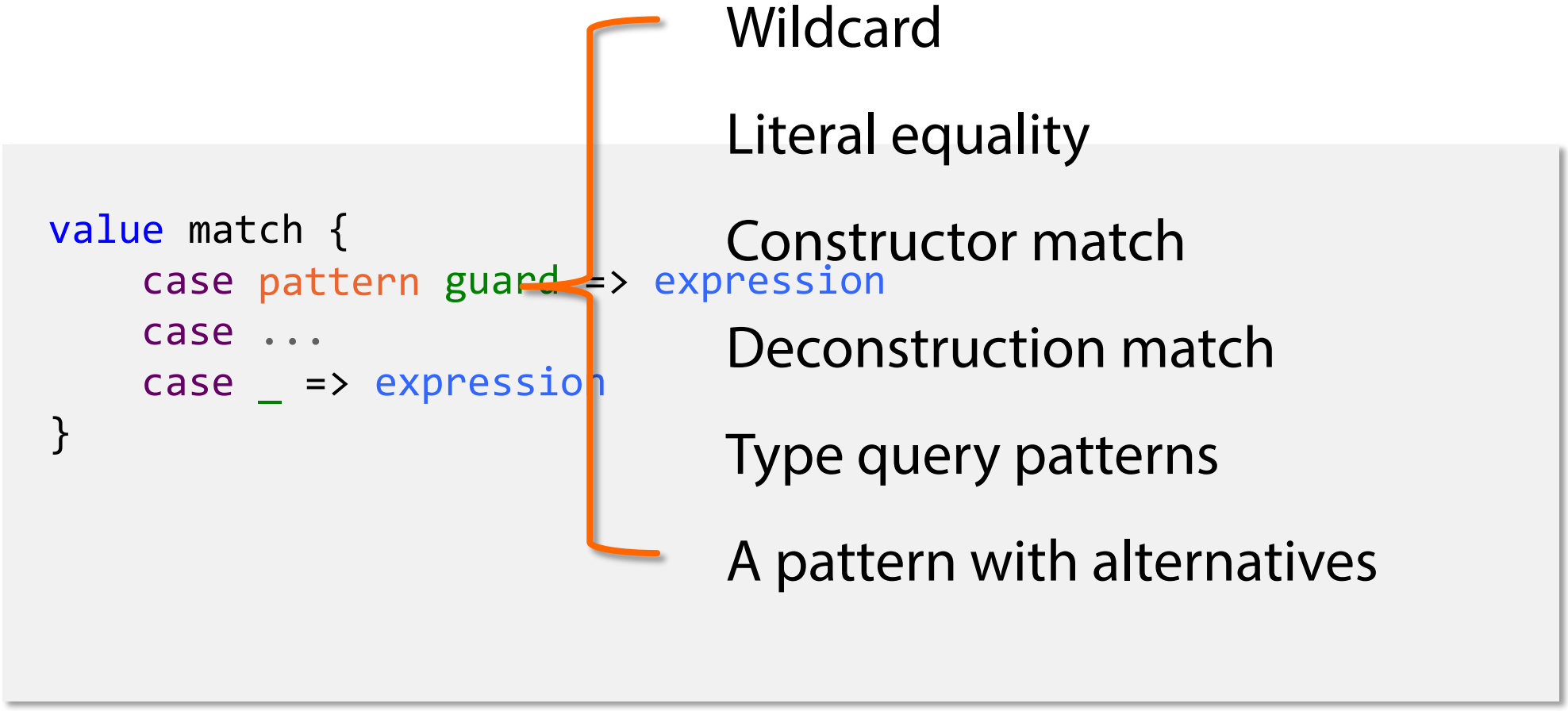
```
value match {
    case ...
    case ...
    case ...
}
```

```
value match {
    case pattern
    case ...
    case ...
}
```

```
value match {
    case pattern guard
    case ...
    case ...
}
```

```
value match {
    case pattern guard => expression
    case ...
    case ...
}
```

```
value match {
    case pattern guard => expression
    case ...
    case _ => expression
}
```


value match {
case pattern guard => expression
case ...
case _ => expression
}
Wildcard
Literal equality
Constructor match
Deconstruction match
Type query patterns
A pattern with alternatives

```
value match {
    case pattern guard => expression
    case ...
    case _ => expression
}
```

```
value match {
    case x => println(x)
    case ...
    case _ => expression
}
```

www.scala-lang.org/files/archive/spec/2.11/08-pattern-matching.html

Literal Matches

demos/01-literal-match.demo

DEMO

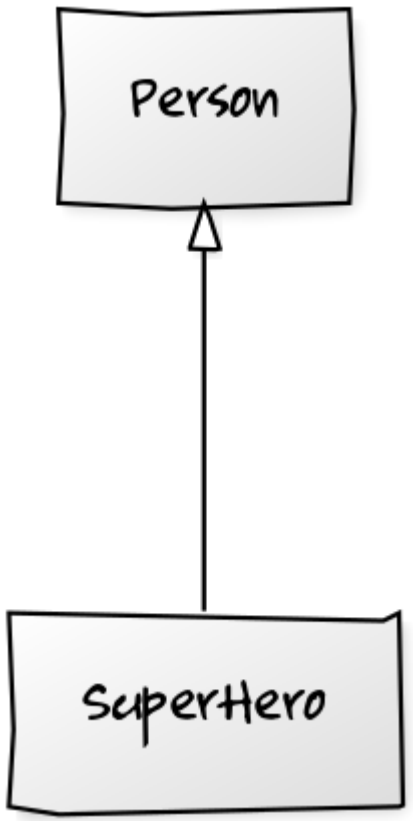
Constructor Matches

Constructor Patterns –

Allow you to match on the arguments used to construct an object.

demos/02-super-hero-constructor-pattern.demo

DEMO


Person
SuperHero

What super-powers does an otherwise unknown person have, if they have an alter-ego of Bruce Wayne?

unapply

Type Queries

demos/03-super-heo-type-query.demo

DEMO


instanceof

Exceptions use Type Queries

demos/04-exceptions.demo

DEMO

Deconstruction Matches

(Extractors)

apply

```
apply(a, b)
```

```
apply(a, b) →
```

```
apply(a, b) → object(a, b)
```

```
apply(a, b) → object(a, b)
```

unapply

```
apply(a, b) → object(a, b)
```

```
unapply(object(a, b))
```

```
apply(a, b) → object(a, b)

unapply(object(a, b)) →
```

```
apply(a, b) → object(a, b)
```

```
unapply(object(a, b)) → a, b
```

Extractors –

Given an object, an extractor typically extracts the parameters that would have created that object

demos/05-unapply.demo.demo

DEMO

www.scala-lang.org/files/archive/spec/2.11/08-pattern-matching.html

Why write your own Extractors?

- Custom Behaviour
- Can’t use a Case Class
- Can’t Modify (e.g. `String`)

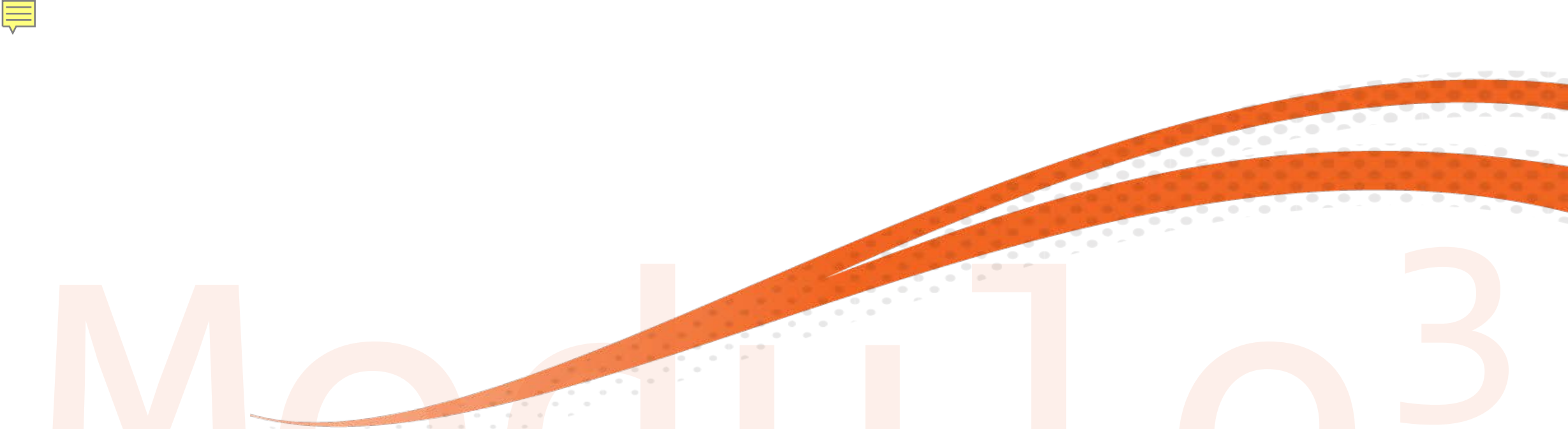
demos/06-url-extractor.demo

DEMO

Representation Independence

Type Queries

Summary

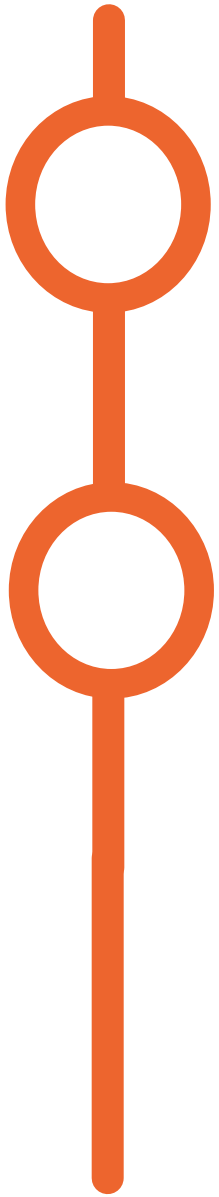


Beyond Java to Scala

Functional Scala: `map` and `flatMap`

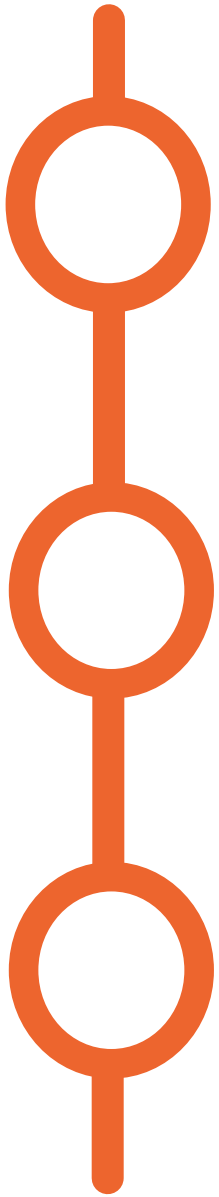
- `map`
- `flatMap`

map, flatMap



map, flatMap

Monads

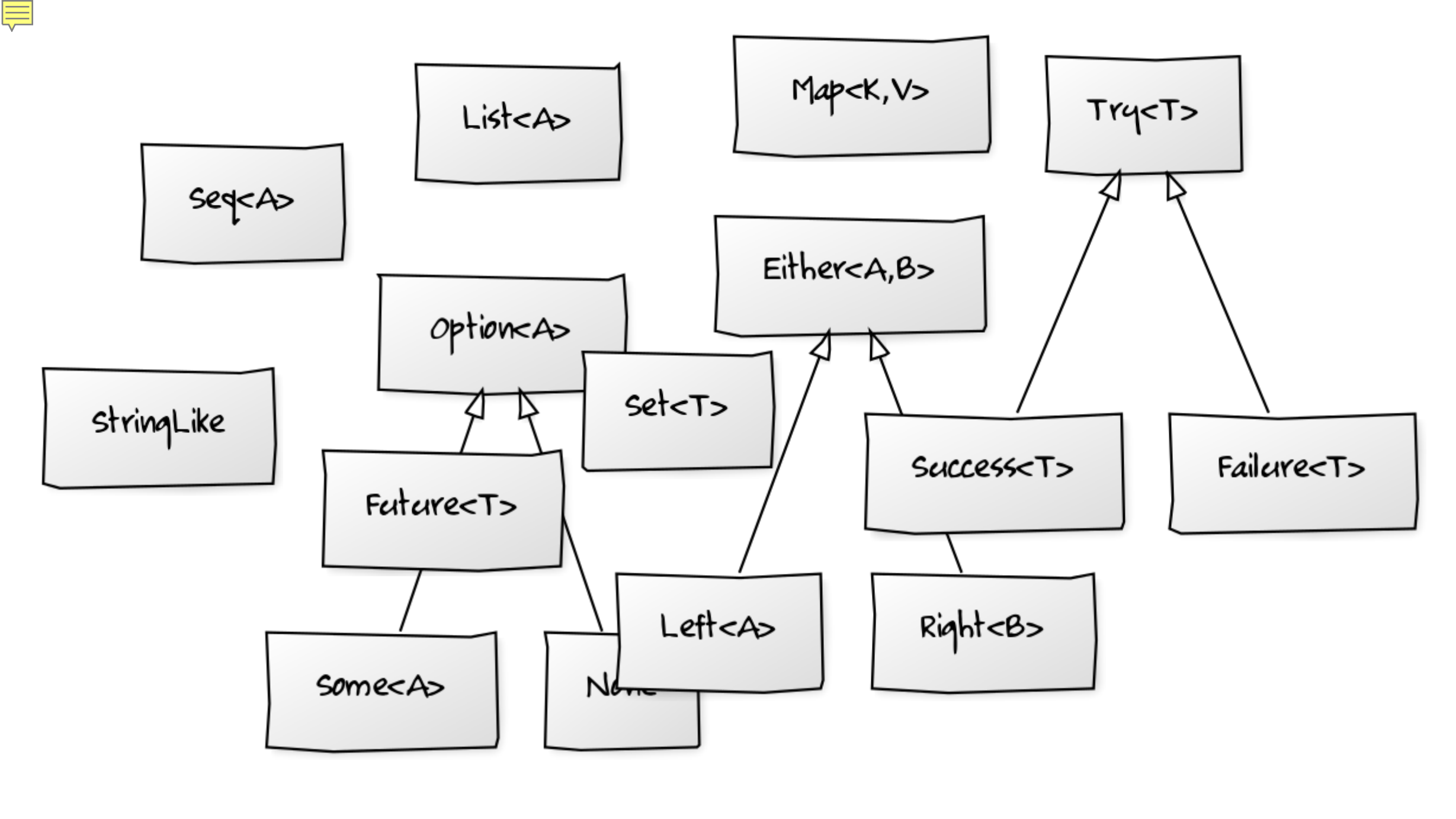


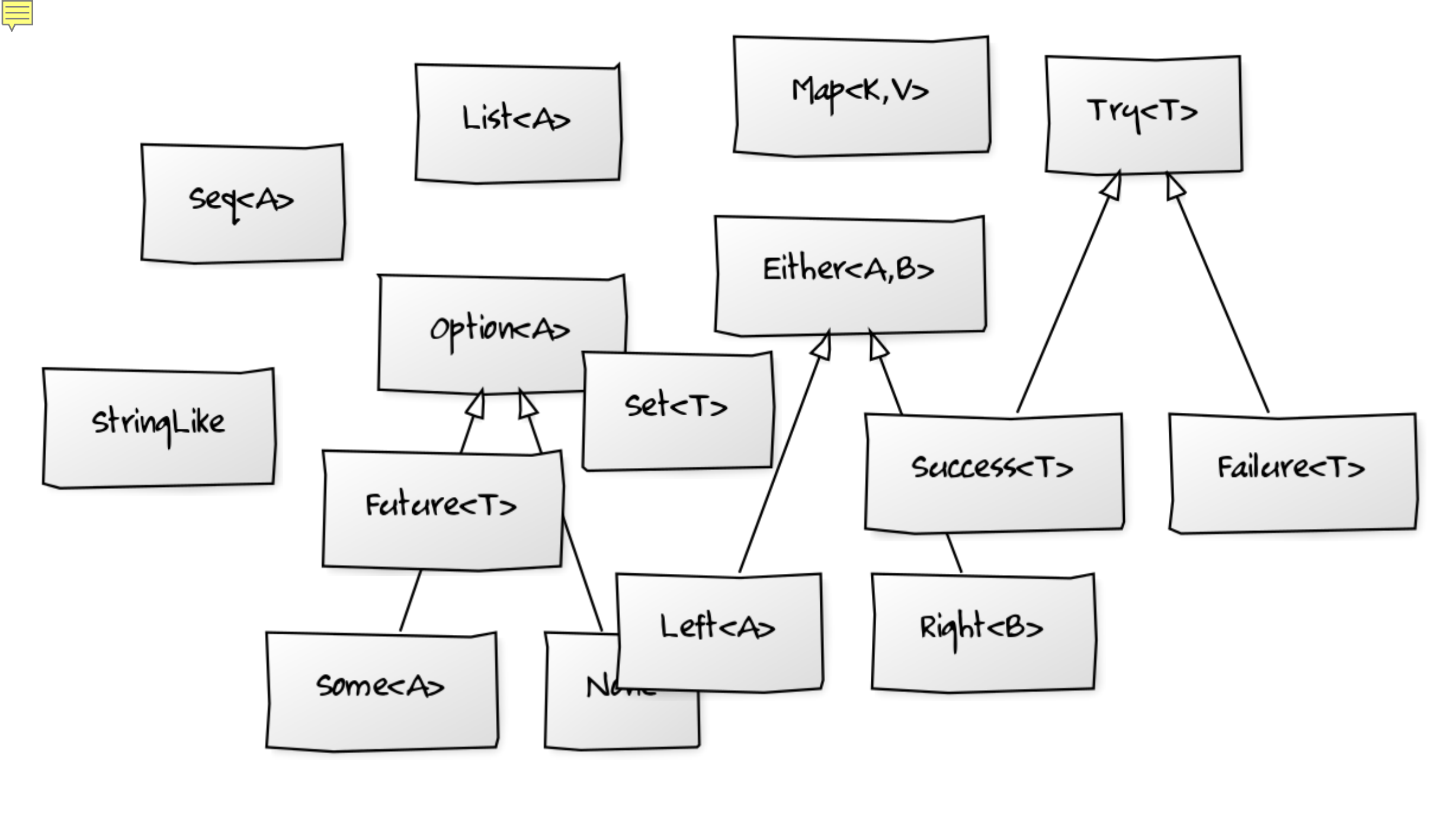
map, flatMap

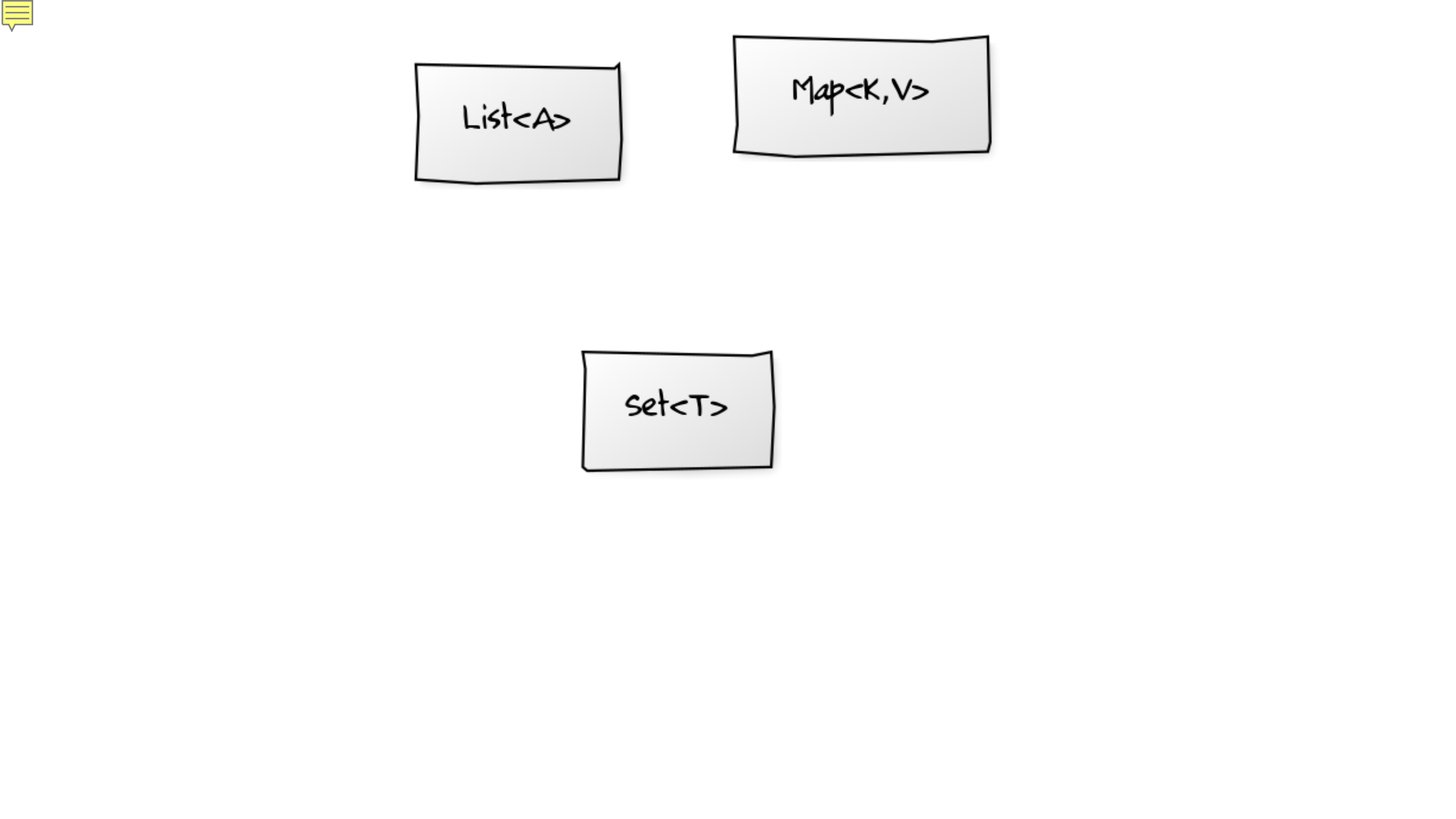
Monads

For-comprehension

The `map` Function


List<A>
Map<K,V>
Try<T>
Seq<A>
Either<A,B>
Option<A>
StringLike
Set<T>
Future<T>
Success<T>
Failure<T>
Left<A>
Right<B>
Some<A>


List<A>
Map<K,V>
Try<T>
Seq<A>
Either<A,B>
Option<A>
StringLike
Set<T>
Future<T>
Success<T>
Failure<T>
Left<A>
Right<B>
Some<A>


List<A>
Map<K,V>
Set<T>


```
def double(x: Int) = x * 2
```

```
val numbers = List(1, 2, 3)
```

```
List(2, 4, 6)
```

demos/01-map-repl.demo

DEMO

`map`  Transforms


map
Like foreach
with extras

demos/02-map-long-hand.demo

DEMO

The `flatMap` Function


with extras
flapMap
Like map

with extras

- Function applies one-to-many transformation
- Flattens `List[List[A]]` to `List[A]`

```
map(f: A => B): List[B]

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

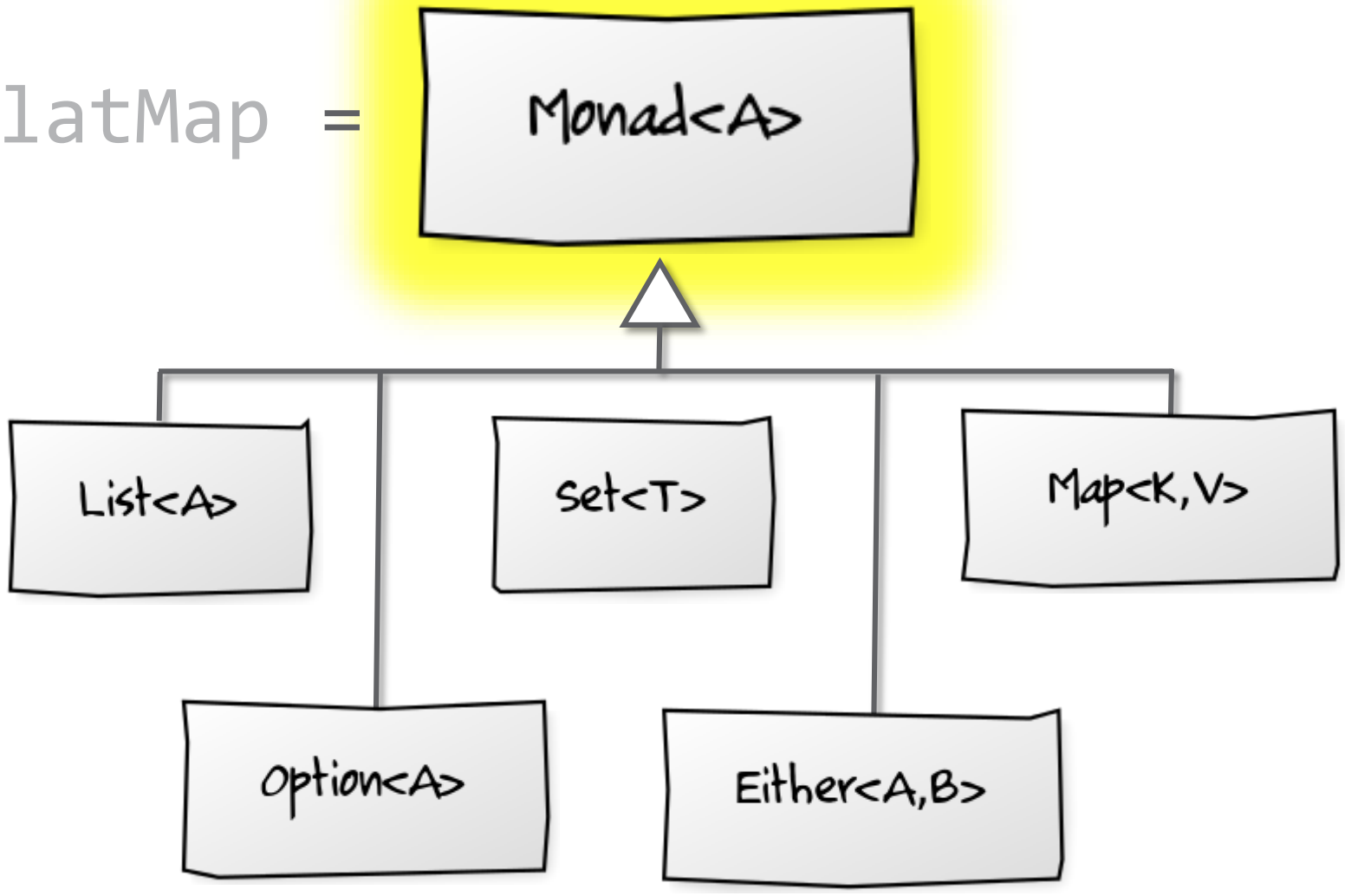
demos/01-map-repl.demo (continued)

DEMO

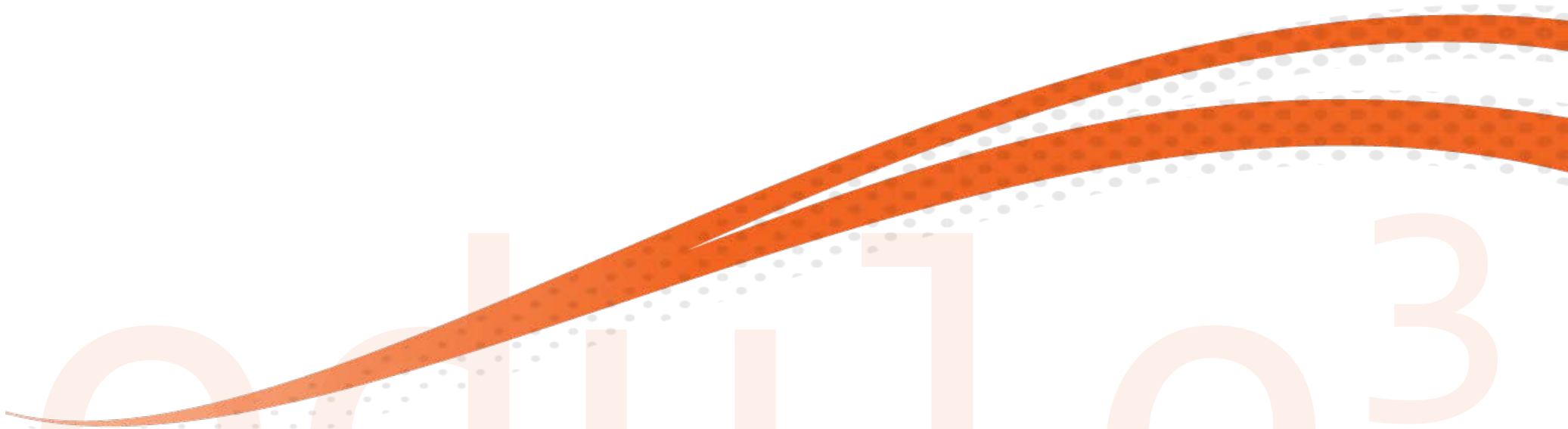
demos/03-flatmap-long-hand.demo

DEMO

map, flatMap =



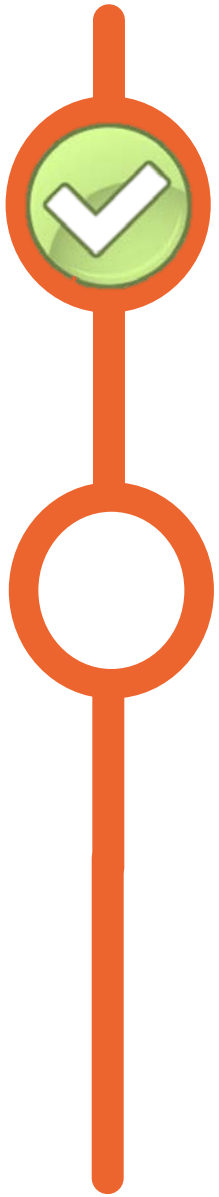
Next up…



Beyond Java to Scala

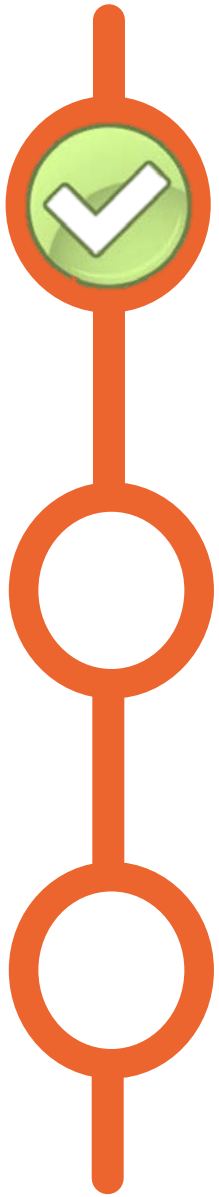
Functional Scala: Monads

map, flatMap



map, flatMap

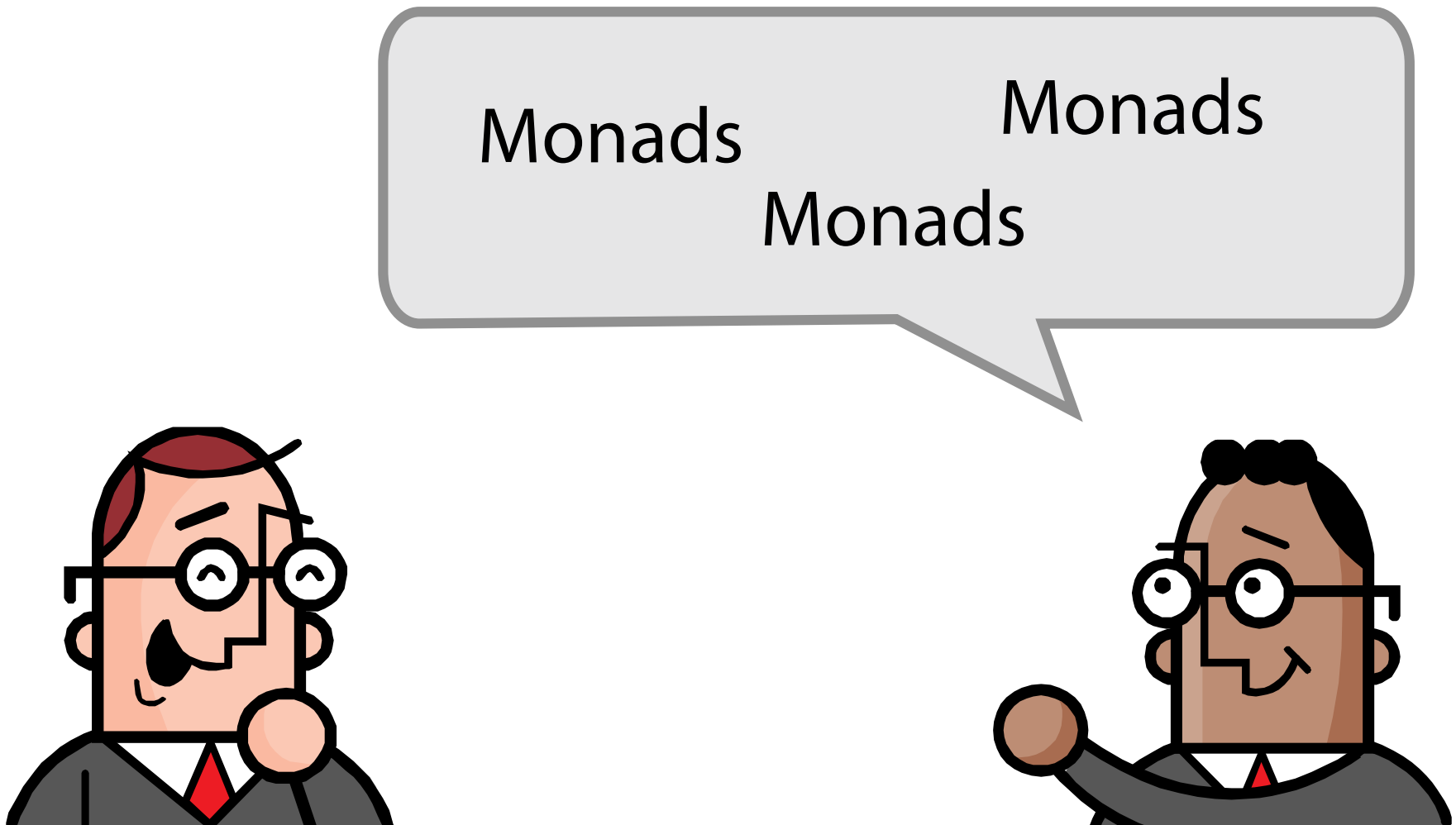
Monads

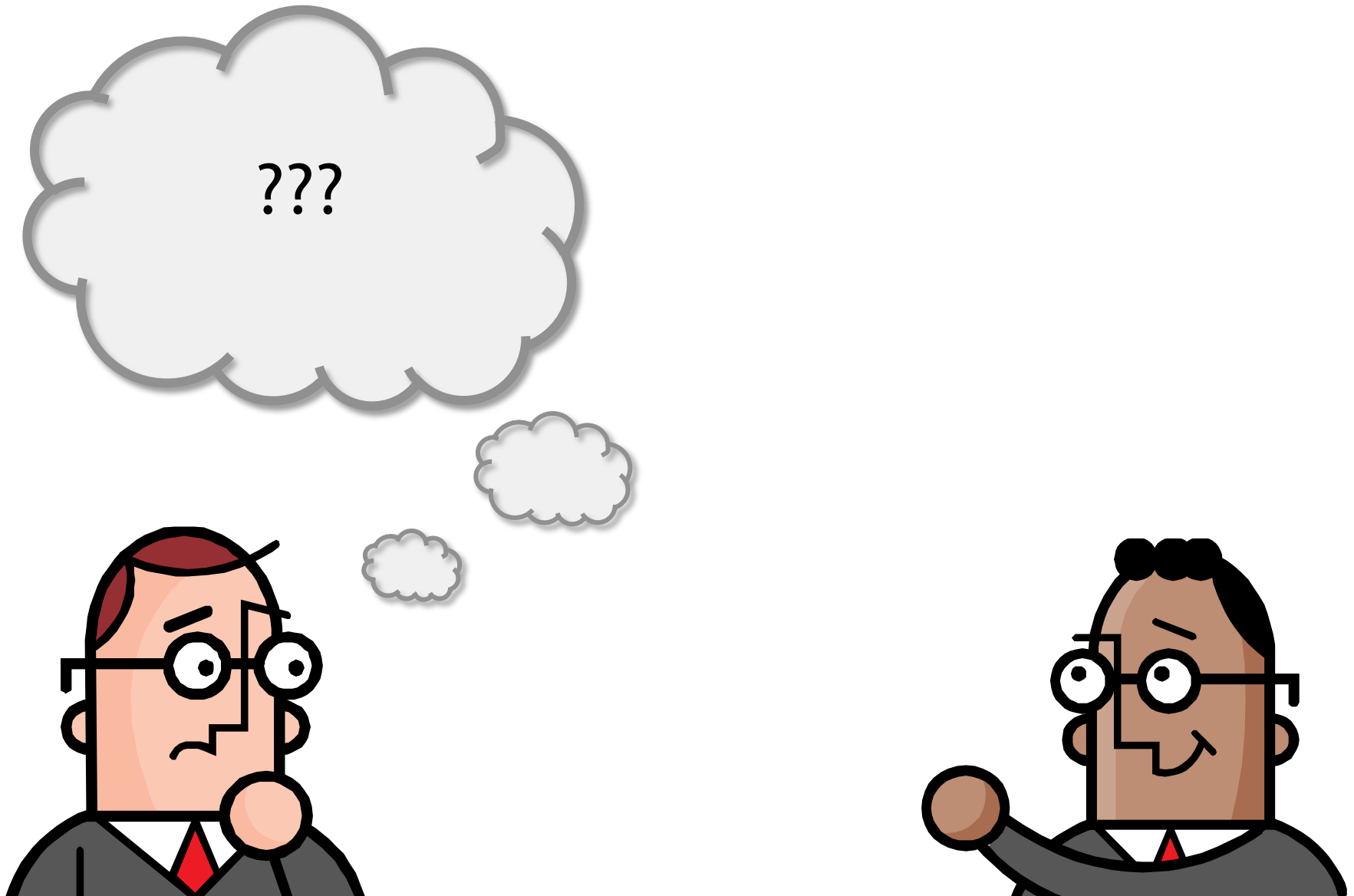


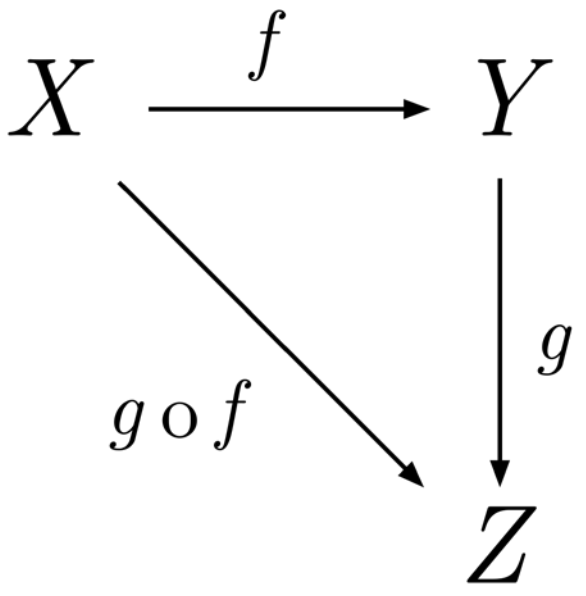
map, flatMap

Monads

For-comprehension


Monads
Monads
Monads




X
f
Y
g
g o f
Z

You

You don’t

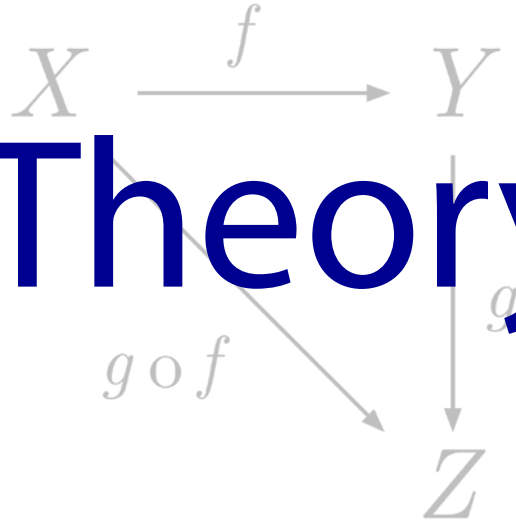
You don’t need

You don’t need to

You don’t need to understand

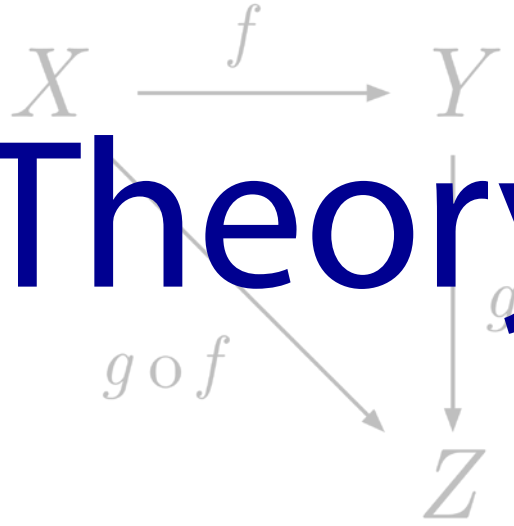
You don’t need to understand

Category Theory



You don’t need to understand

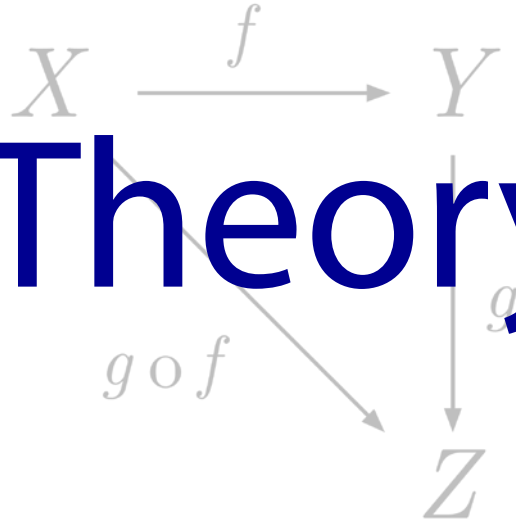
Category Theory



for

You don’t need to understand

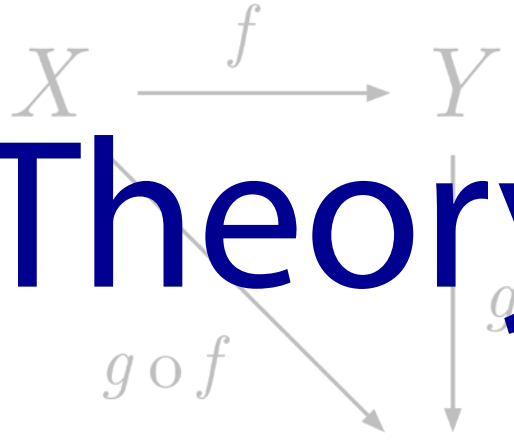
Category Theory



for functional

You don’t need to understand

Category Theory



for functional programming

You don’t need to understand

Haskell

You don’t need to understand

Haskell

to

You don’t need to understand

Haskell

to program

You don’t need to understand

Haskell

to program with

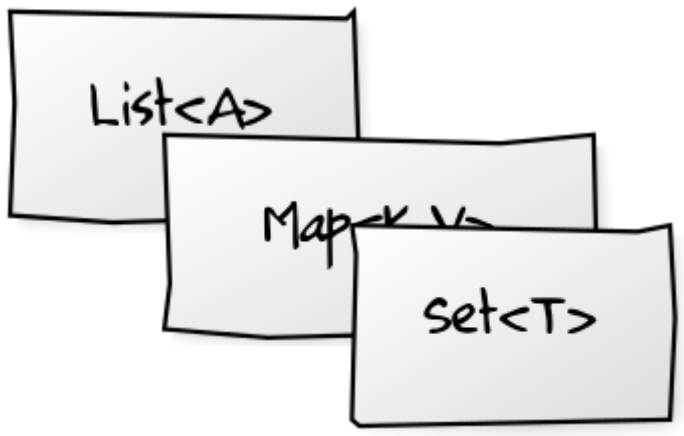
You don’t need to understand

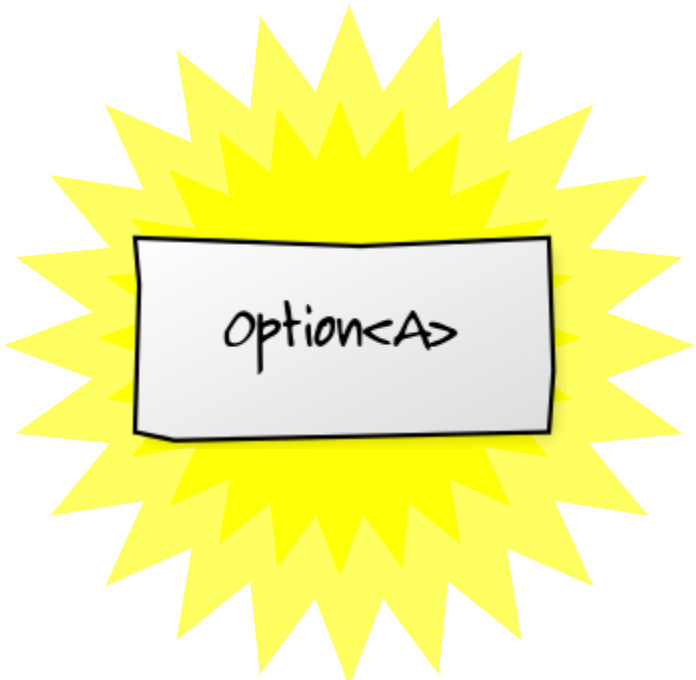
Haskell

to program with Scala

Layman's Definition –

Something that has `map` and `flatMap` functions.


List<A>
Map<K,V>
Set<T>

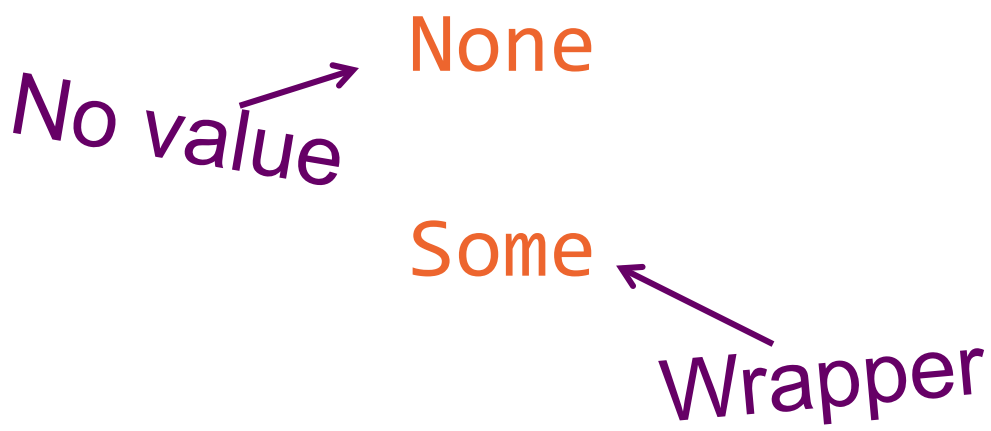

Option<A>

How does `Option` avoid nulls?

What's `Option` got to do with Monads?

How does `Option` avoid nulls?

Return a type of `Option` to represent



What's `Option` got to do with Monads?

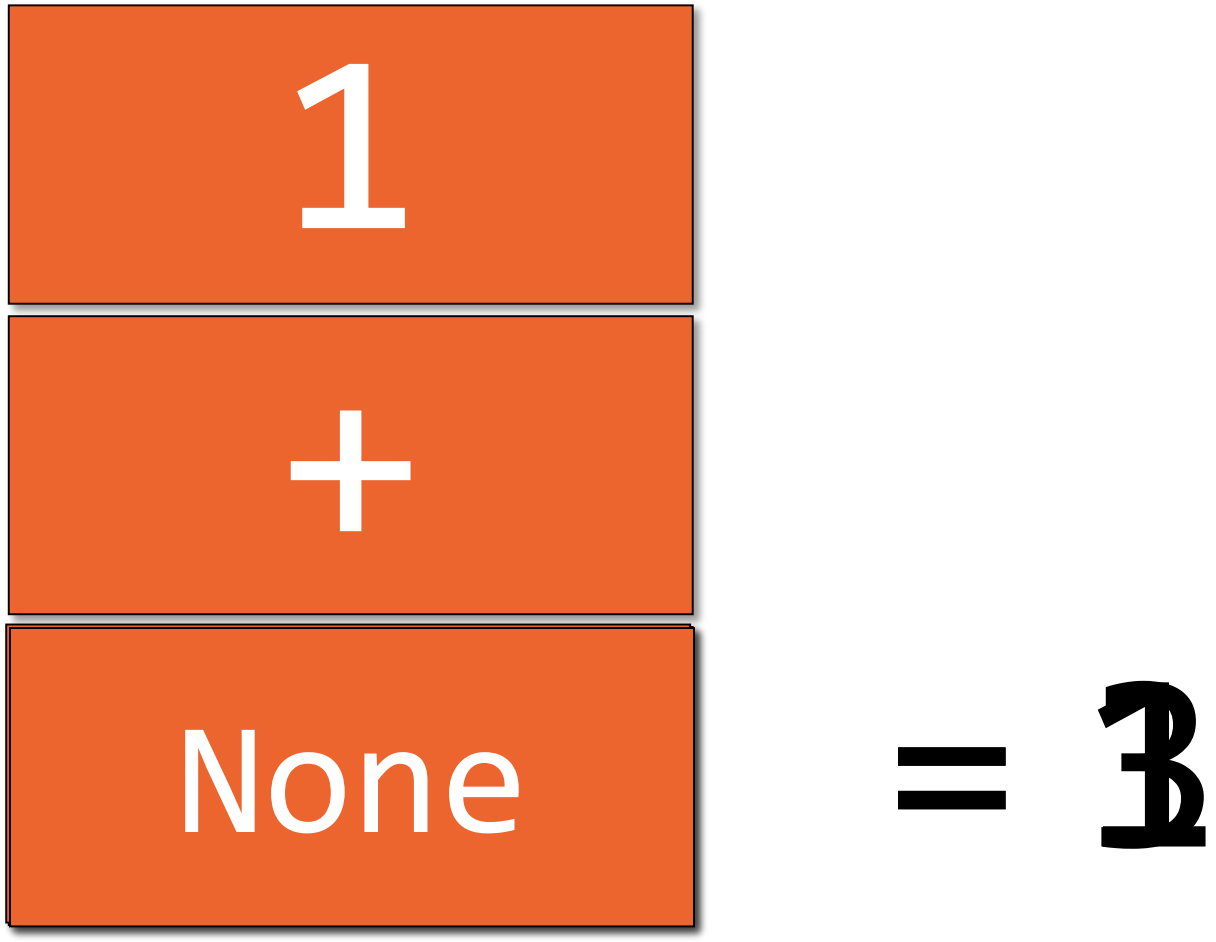
You treat the option consistently using

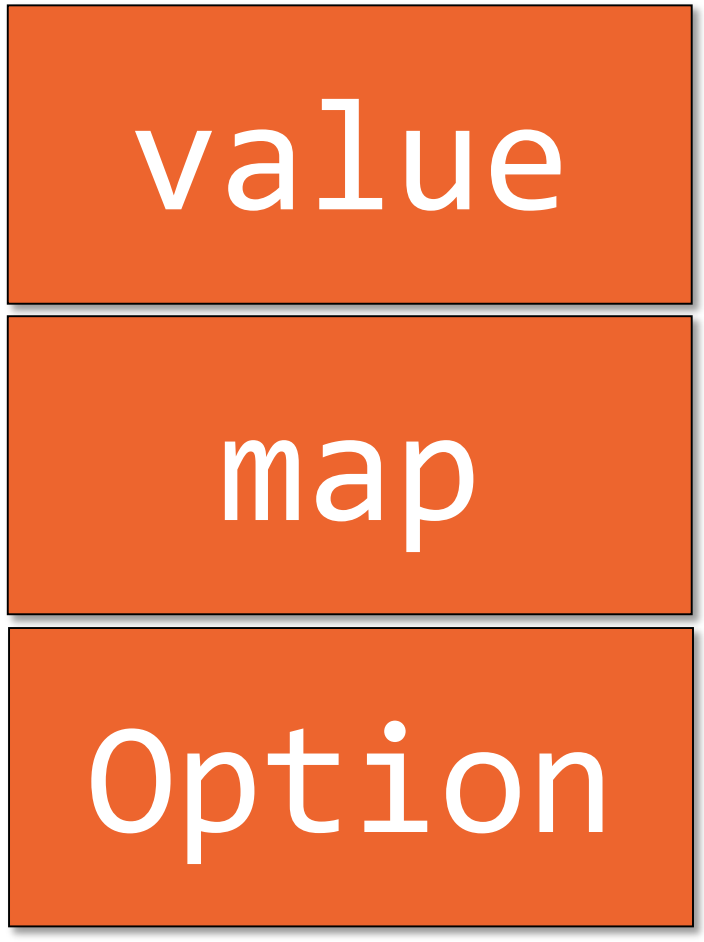
`map`

`flatMap`

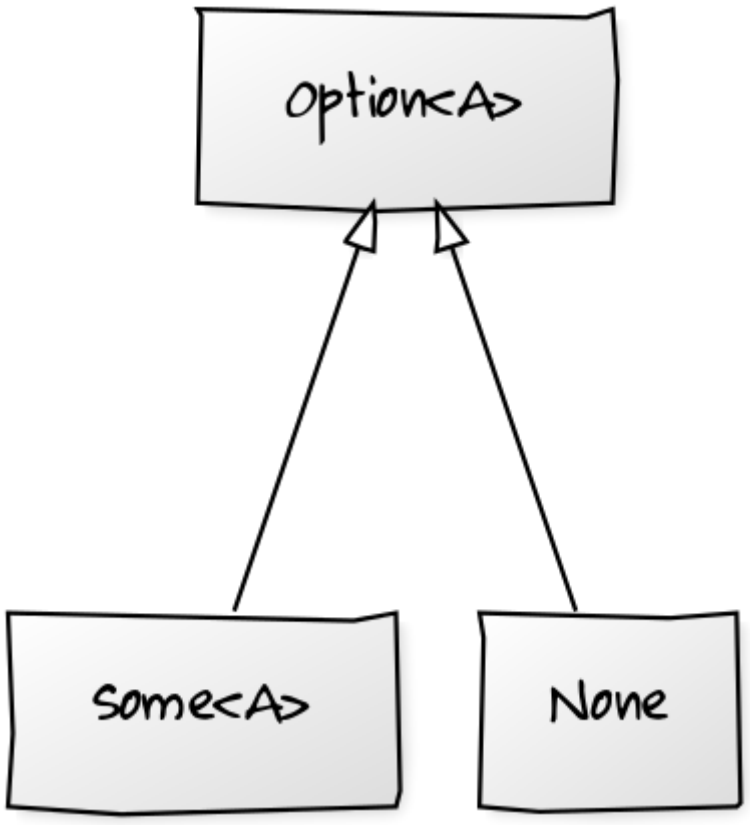
Therefore, it is a monad

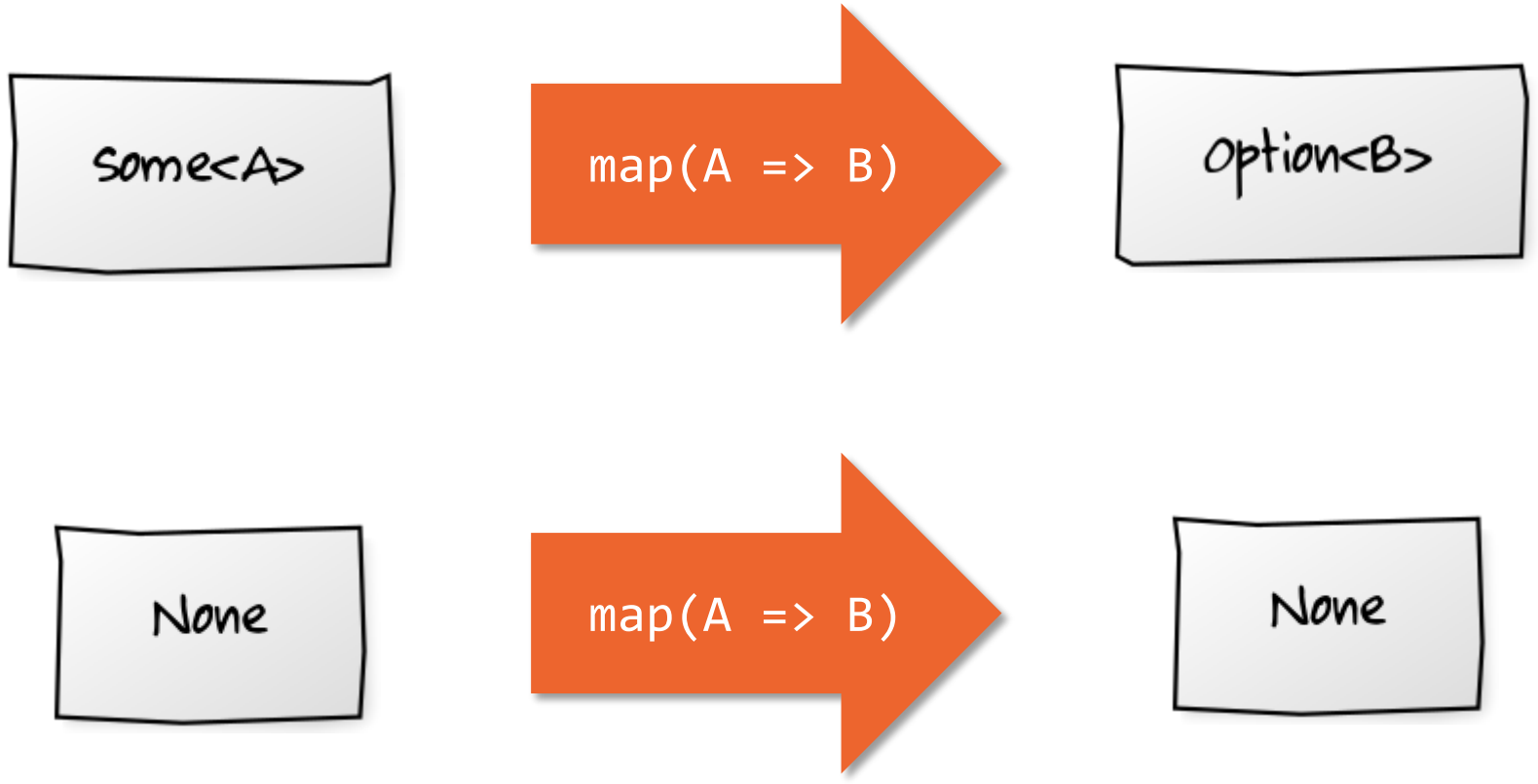
Null Object Pattern


1
+
None
= 3

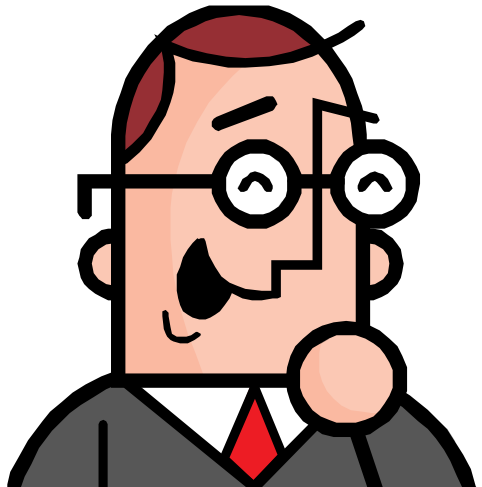

value
map
Option

The `map` function


Option<A>
Some<A>
None


Some<A>
map(A => B)
Option<B>
None
map(A => B)
None

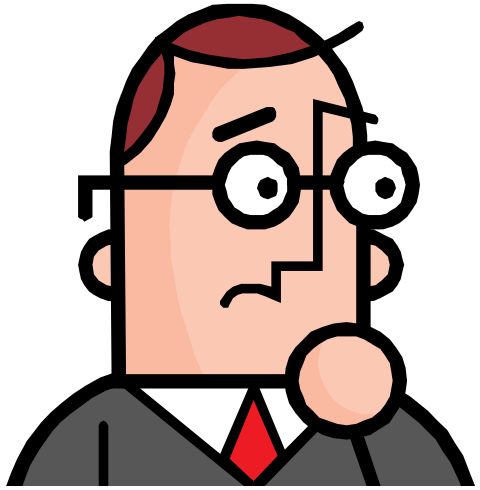
Avoid Null Checks



demos/01-customers-null.demo

DEMO

- Communicating Intent
- Forcing Unhappy Path Handling



demos/02-customers-option.demo

DEMO


LEVEL UP!

The `flatMap` function

Monad Definition –

1. Operate on a parameterised type
2. Be able to construct itself from that type
3. Provide a `flatMap` function

	Option	List
Parameterised type	`Option[A]`	`List[T]`

Construction Signature	`Option.apply(x)`
	`Some(x)`
	`None`
		`List(x, y, z)`

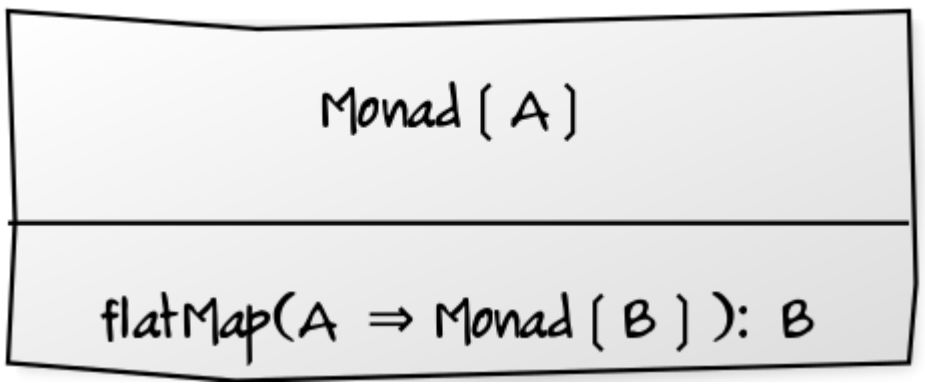
flatMap Signature	`def flatMap[B]`
	`  (f: A => Option[B]):`
	`  Option[B]`
		`def flatMap[B]`
		`  (f: A => List[B]):`
		`  List[B]`

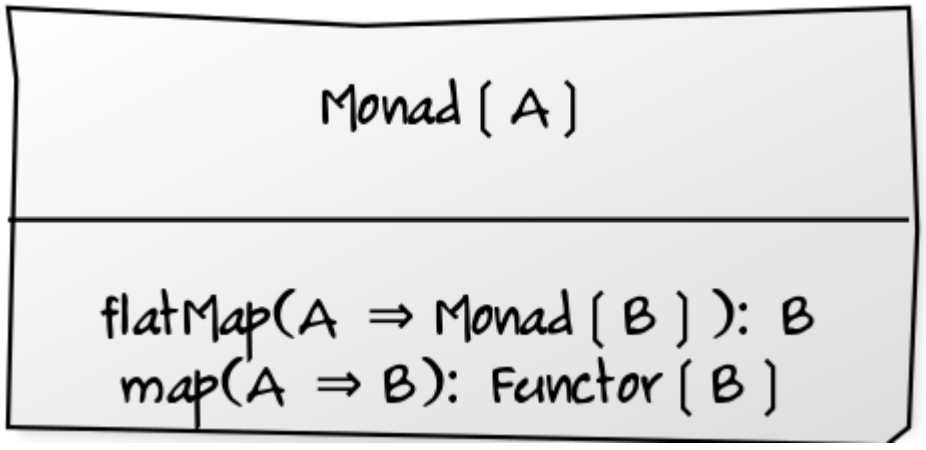
Layman's Definition –

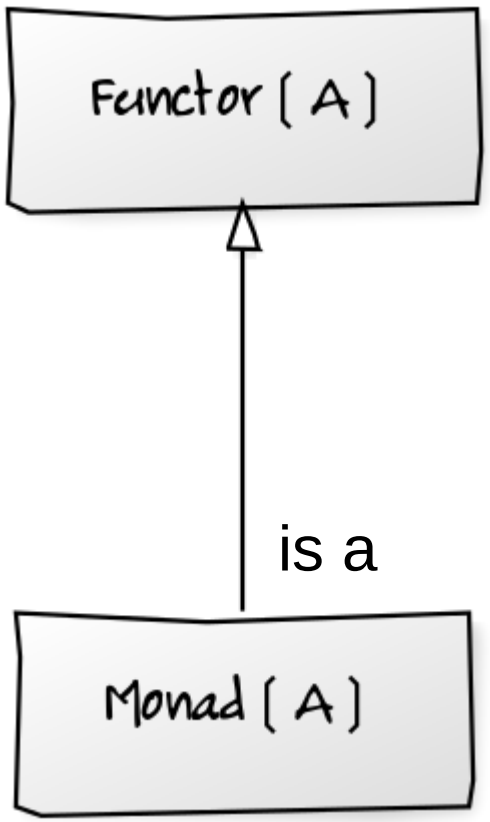
Something that has `map` and `flatMap` functions.

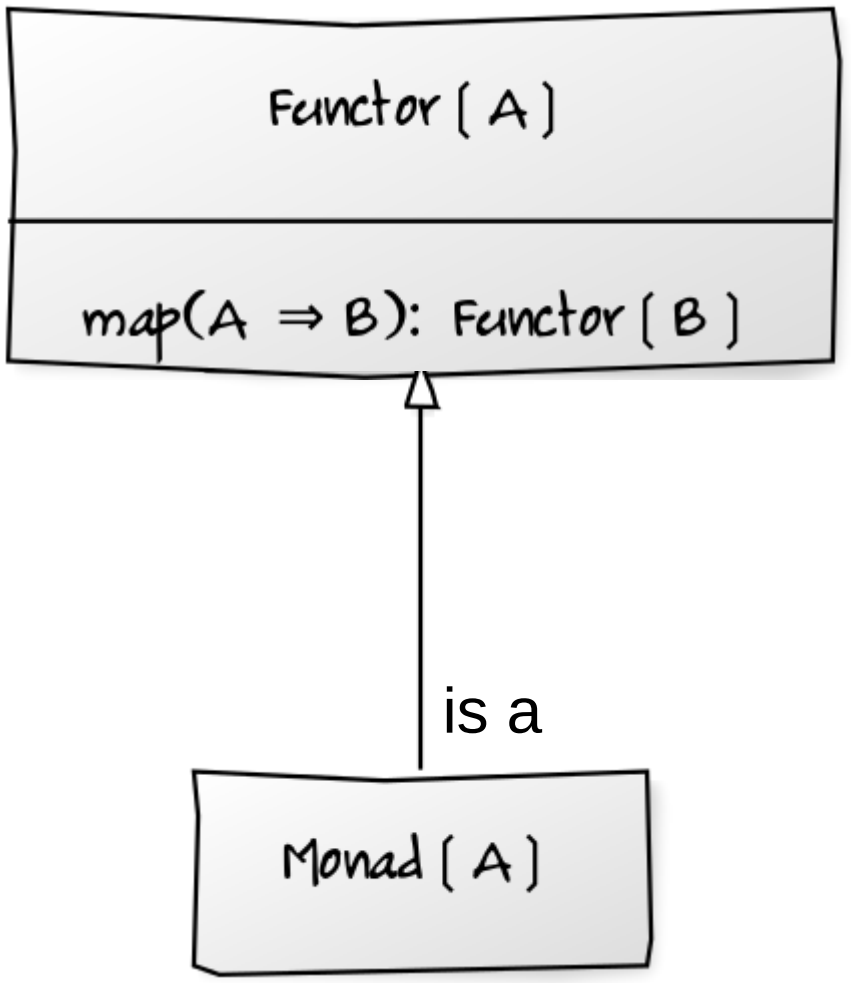
What Gives?


Monad [A]


Monad [A]
flatMap(A ⇒ Monad [B]): B


Monad [A]
flatMap(A ⇒ Monad [B]): B
map(A ⇒ B): Functor [B]


Functor [A]
is a
Monad [A]


Functor [A]
map(A ⇒ B): Functor [B]
is a
Monad [A]

Summary

Monadic Behaviour

`map`

`flatMap`

PWR UP

Differ by `Type`

What is a Monad?

map, flatMap, Monads

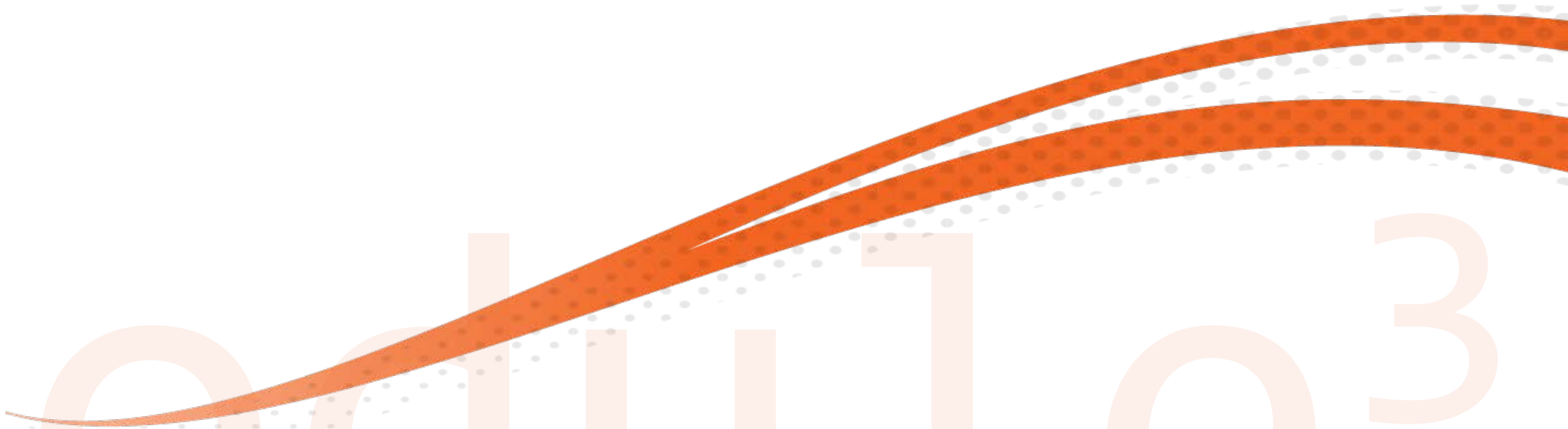
More Monads

For-comprehension

map, flatMap, Monads

More Monads

For-comprehension



Beyond Java to Scala

Functional Scala: For-Comprehensions In-Depth

- Chaining Monadic Functions
- Yielding Results
- For-Comprehensions
- Under the Covers

map, flatMap

map, flatMap

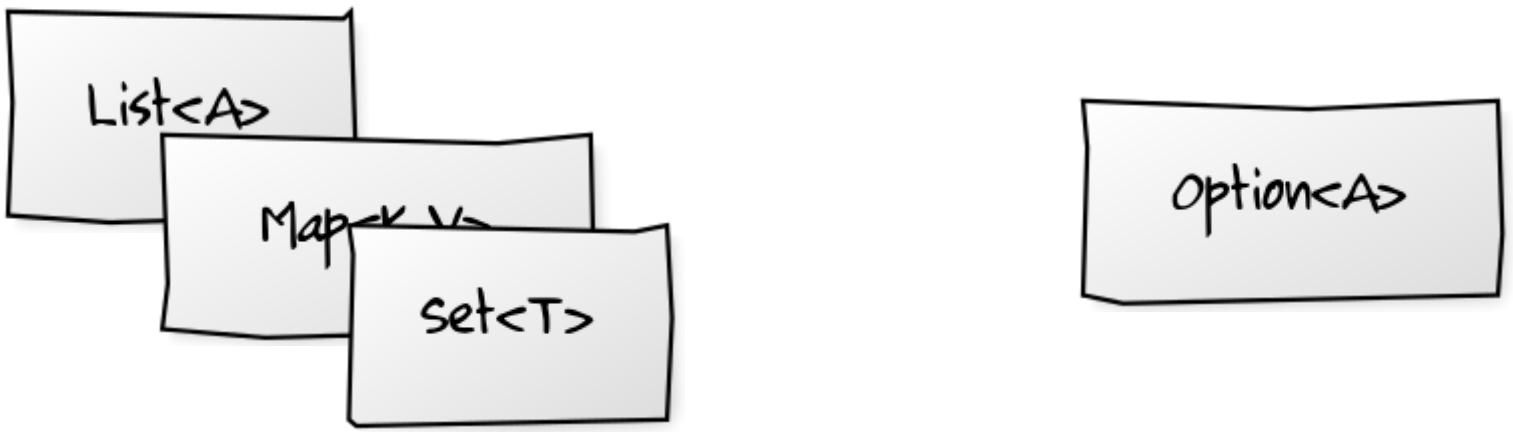
Monads

map, flatMap

Monads

For-comprehension

flatMap


List<A>
Map<K,V>
Set<T>
Option<A>

```
val usCustomers = Set("Albert", "Beatriz”, ...)

usCustomers.flatMap(name =>
 customers.find(name)).map(customer => customer.total).sum
```


P
USPS PRIORITY MAIL
BELTSVILLE MD
U.S. POSTAGE
$3.95
Sample Mailer
1123 Main St
Test City DC 20260
ADDRESS SERVICE REQUESTED
SHIP TO:
Martin Odersky
EPFL,
SWITZERLAND
e/ USPS SIGNATURE CONFIRM
9121 0268 3733 1000 0010 10
ELECTRONIC RATE APPROVED #026837331
Priority Mail is a registered trademark of the U. S. Postal Service.

1

A customer may or may not exist in the repository

2

A customer may or may not exist in the repository

A customer may or may not have an address

3

A customer may or may not exist in the repository

A customer may or may not have an address

An address must have a street but may not have a postal code

1

```
customer = customers.find(name)
```

2

```
customer = customers.find(name)

address = customer.address
```

3

```
customer = customers.find(name)

address = customer.address

street = address.street
postcode = address.postcode
```

3

Could be null

```
customer = customers.find(name)

address = customer.address
street = address.street
postcode = address.postcode
```

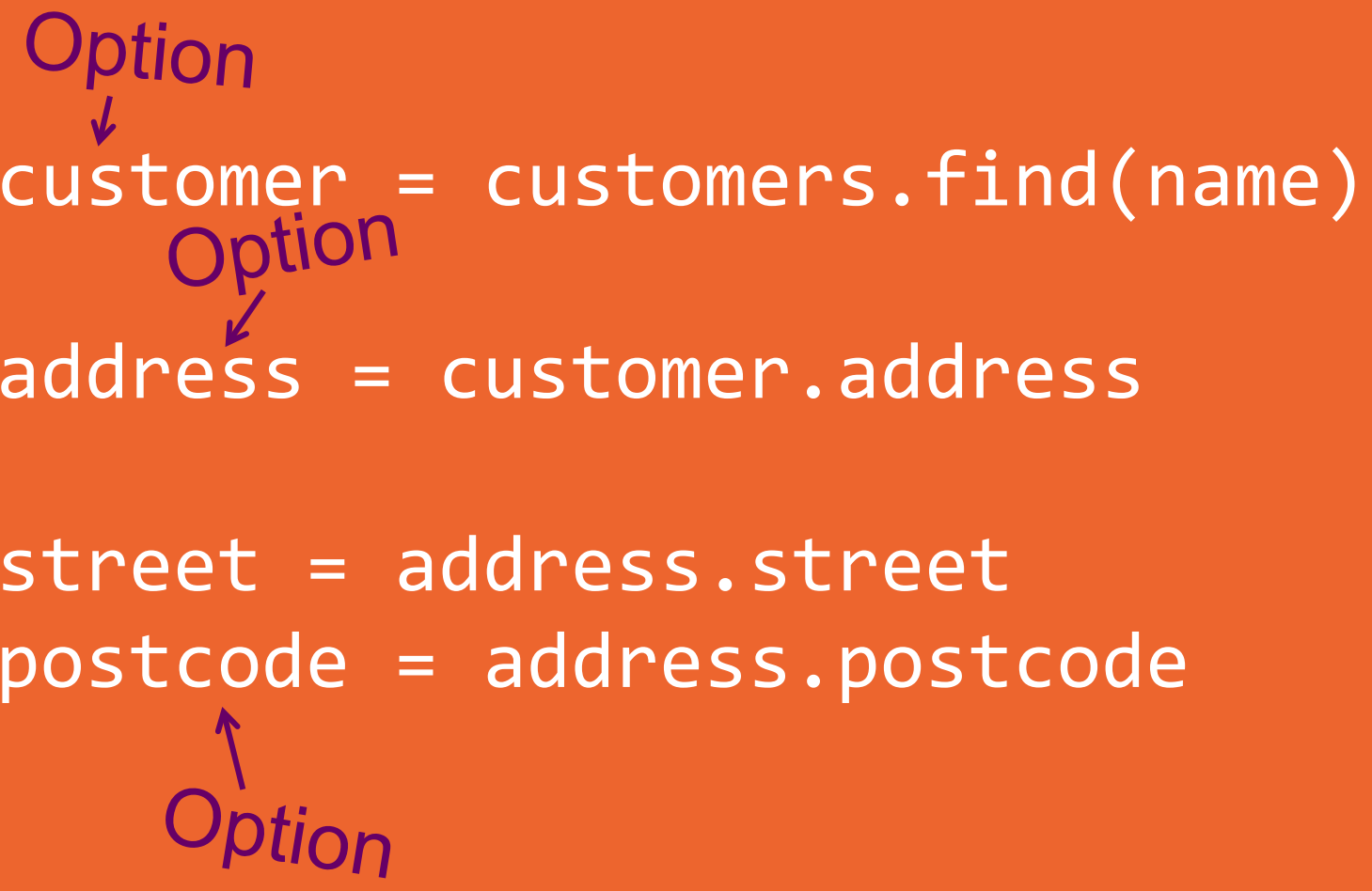
Could be null

Could be null

demos/01-generate-label-null.demo

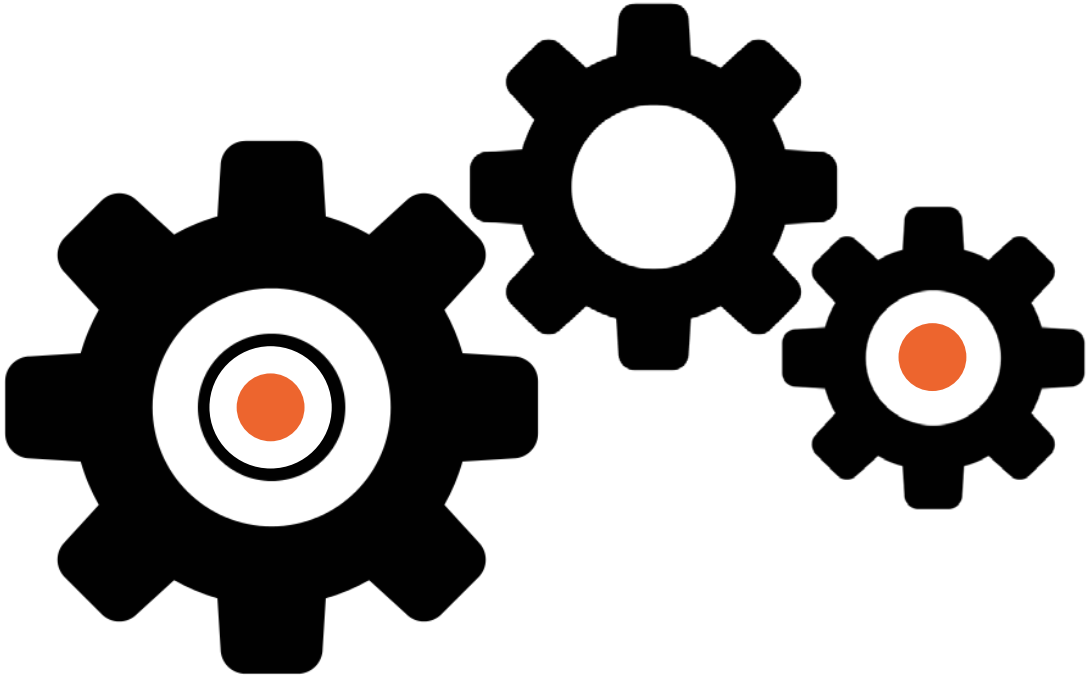
DEMO

3


Option
customer = customers.find(name)
Option
address = customer.address
street = address.street
postcode = address.postcode
Option

demos/02-generate-label-option.demo

DEMO



How For-Comprehensions Work

for

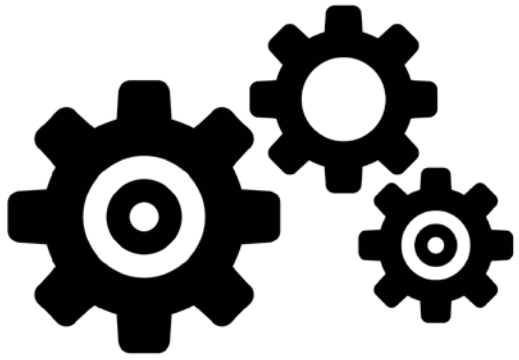
for (nested)

```
def foreach(f: A => B): Unit
```

for with yield

- Conditional Semantics
- Extensible
- Declarative
- For all Monads

Summary



flatMap
.flatMap
.flatmap

for => foreach
for/yield => map
Nested yields => flatMap/map


That's all folks!
For module 3

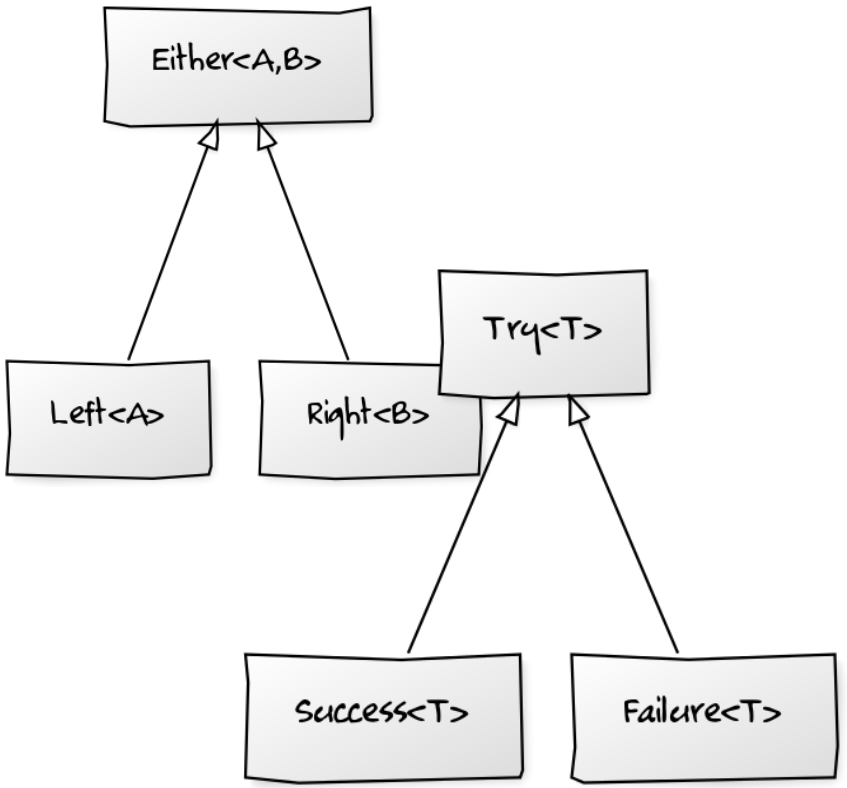
Further Reading

- Case Classes

- Case Classes
- Type Aliases / Aliasing Imports

</XML>

- Case Classes
- Type Aliases / Aliasing Imports
- Native XML Support



- Case Classes
- Type Aliases / Aliasing Imports
- Native XML Support
- `Either` / `Try`

- Case Classes
- Type Aliases / Aliasing Imports
- Native XML Support
- `Either` / `Try`
- Tail Recursion

- Case Classes
- Type Aliases / Aliasing Imports
- Native XML Support
- `Either` / `Try`
- Tail Recursion
- Implicits

- Case Classes
- Type Aliases / Aliasing Imports
- Native XML Support
- `Either` / `Try`
- Tail Recursion
- Implicits
- More…

Next Up…