Some(slides)

```
val reasonsToUseNull = None
```

Who am I?

- Java (& Scala) Developer at Schantz A/S
- Polyglot curious, Coursera junkie
- Interested in HCI and Usability
- https://github.com/JKrag
- @jankrag
- Geek, builder and flyer of kites, reptile & cat breeder, Rubik's puzzle fan

Oh we wish...

```
val customer = Customers.findById(1234)
customer.getAccount(FUNSTUFF).getLastInterest.getAmount
```

Oh we wish...

```
val customer = Customers.findById(1234)
customer.getAccount(FUNSTUFF).getLastInterest.getAmount
```

NullPointers !

Classic solutions (java)

Classic solutions (java)

Nested if's

```
if(customer != null {
   if(customer.getAccount(FUNSTUFF) != null) {
      if(customer.getAccount(FUNSTUFF).getLastInterest != null) {
         return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
      }
   }

}
return null;
```

Classic solutions (java)

Nested if's

```
if(customer != null {
   if(customer.getAccount(FUNSTUFF) != null) {
      if(customer.getAccount(FUNSTUFF).getLastInterest != null) {
         return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
      }
   }

}
return null;
```

UGLY

Classic solutions (java)

Nested if's

```
if(customer != null {
   if(customer.getAccount(FUNSTUFF) != null) {
      if(customer.getAccount(FUNSTUFF).getLastInterest != null) {
         return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
      }
   }

}
return null;
```

UGLY

Early returns

```
if (customer == null) return null;
if (customer.getAccount(FUNSTUFF) == null) return null;
if (customer.getAccount(FUNSTUFF).getLastInterest == null) return null;
return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
```

Classic solutions (java)

Nested if's

```
if(customer != null {
   if(customer.getAccount(FUNSTUFF) != null) {
      if(customer.getAccount(FUNSTUFF).getLastInterest != null) {
         return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
      }
   }

}
return null;
```

UGLY

Early returns

```
if (customer == null) return null;
if (customer.getAccount(FUNSTUFF) == null) return null;
if (customer.getAccount(FUNSTUFF).getLastInterest == null) return null;
return customer.getAccount(FUNSTUFF).getLastInterest.getAmount
```

still UGLY!

Same in Scala

Same in Scala

```scala
val customer = Customers.findById(1234)
if (customer != null) {
   val account = customer.account(FUNSTUFF);
   if (account != null) {
      val interest = account.getLastInterest
      if (interest != null)
         interest.amount
      else
         null
   } else
      null
} else
   null
```

Same in Scala

```
val customer = Customers.findById(1234)
if (customer != null) {
   val account = customer.account(FUNSTUFF);
   if (account != null) {
      val interest = account.getLastInterest
      if (interest != null)
         interest.amount
      else
         null
   } else
      null
} else
   null
```

Even in Scala, Still UGLY!

Same in Scala

```
val customer = Customers.findById(1234)
if (customer != null) {
   val account = customer.account(FUNSTUFF);
   if (account != null) {
      val interest = account.getLastInterest
      if (interest != null)
         interest.amount
      else
         null
   } else
      null
} else
   null
```

Even in Scala,
Still Ugly!

and error prone

non-existence

non-existence

Java

null, null, null, null :-(

non-existence

Java

null, null, null, null :-(

Groovy (et al.)

Safe navigation operator

```
def amount = customer?.account?.interest?.amount
```

non-existence

Java

null, null, null, null :-(

Groovy (et al.)

Safe navigation operator

```
def amount = customer?.account?.interest?.amount
```

Ceylon, Kotlin etc.

both nullable and null-safe types...

```
String name = null; //compile error: null is not an instance of String
String? name = null; //OK
```

non-existence

Java

null, null, null, null :-(

Groovy (et al.)

Safe navigation operator

```
def amount = customer?.account?.interest?.amount
```

Ceylon, Kotlin etc.

both nullable and null-safe types...

```
String name = null; //compile error: null is not an instance of String
String? name = null; //OK
```

Others (e.g. Clojure): **‘nil’ type** - close but...!

non-existence

Java

null, null, null, null :-(

Groovy (et al.)

Safe navigation operator

```
def amount = customer?.account?.interest?.amount
```

Ceylon, Kotlin etc.

both nullable and null-safe types...

```
String name = null; //compile error: null is not an instance of String
String? name = null; //OK
```

Others (e.g. Clojure): **'nil' type** - close but...!

Scala ....

non-existence

Java

null, null, null, null :-(

Groovy (et al.)

Safe navigation operator

```
def amount = customer?.account?.interest?.amount
```

Ceylon, Kotlin etc.

both nullable and null-safe types...

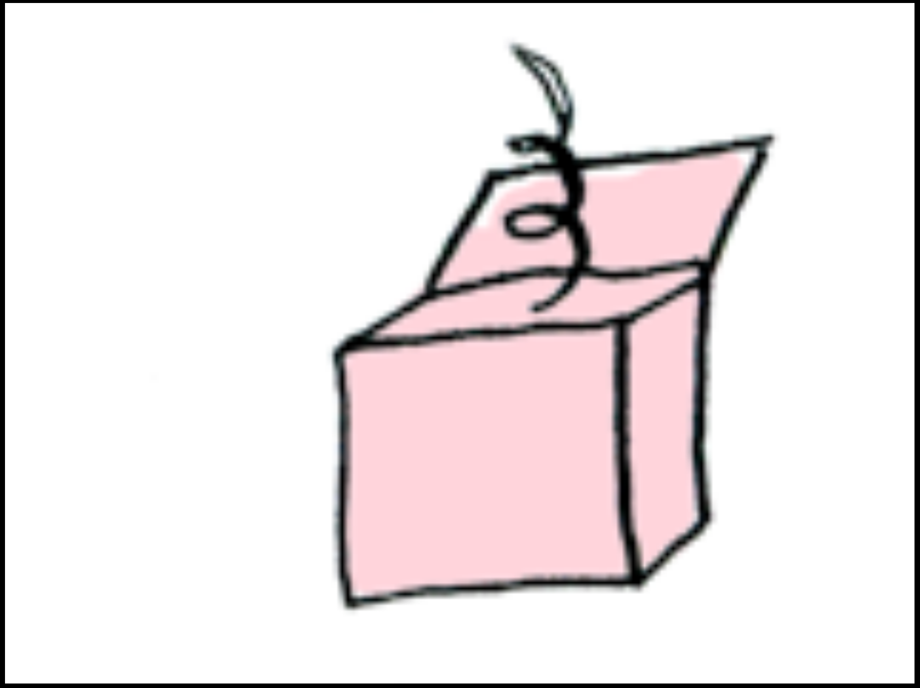
```
String name = null; //compile error: null is not an instance of String
String? name = null; //OK
```

Others (e.g. Clojure): **'nil' type** - close but...!

Scala .... patience...

We need something like:

Container



Empty container

Important: Same ‘shape’ outside

Let me present:

Let me present:

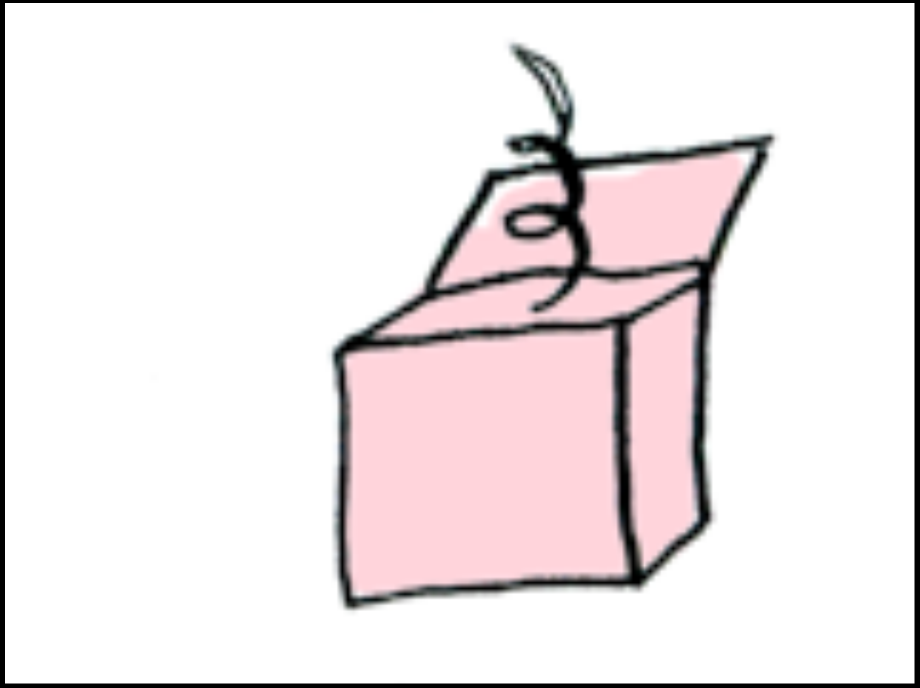
Option monad

Let me present:

Option monad SHHH

Scala's `Option` type:

Some(2)



None

Option - concept

```scala
sealed trait Option[A]

case class Some[A](a: A) extends Option[A]
case class None[A] extends Option[A]
```

Advantages

- Values that may or may not exist now stated in type system
- Clearly shows possible non-existence
- Compiler forces you to deal with it
- You won’t accidentally rely on value

Option - in RL

```scala
sealed abstract class Option[A] extends Product

case class Some[+A](a: A) extends Option[A]

case object None extends Option[Nothing]
```

Option - in RL

```
sealed abstract class Option[A] extends Product {
  def isEmpty: Boolean
  def get: A
 ...
}

final case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

WAT?

Creating Options

Creating Options

- Direct:

```
val o = Some(3)
          //> o : Option[Int] = Some(3)
val n = None
          //> n : None.type = None
```

Creating Options

- Direct:

```
val o = Some(3)
     //> o : Option[Int] = Some(3)
val n = None
     //> n : None.type = None
```

```
BUT NEVER: val aaargh = Some(null)
```

Creating Options

- Direct:

```
val o = Some(3)
     //> o : Option[Int] = Some(3)
val n = None
     //> n : None.type = None
```

```
BUT NEVER: val aaargh = Some(null)
```

- Factory method on companion object:

```
val o = Option(3)
     //> o : Option[Int] = Some(3)
val nn = Option(null)
     //> nn  : Option[Null] = None
```

```scala
val schroedingersBox : Option[Cat] =
   if(random.nextBoolean) then
      Some(Garfield)
   else
      None
```

Many mays to use

- isDefined
- isEmpty

Many mays to use

- isDefined
- isEmpty

```
if (customer.isDefined)
  customer.account;
```

Many mays to use

- isDefined
- isEmpty

```
if (customer.isDefined)
  customer.account;
```

Much more type-safe and null-safe than original null-based java-flavour, but code just as ugly

get?

```
  three.get
  //> res10: Int = 3

nope.get
  //> java.util.NoSuchElementException: None.get
```

get?

```
  three.get
  //> res10: Int = 3

nope.get
  //> java.util.NoSuchElementException: None.get
```

```
 $> Yay. We can still write the other
ugly version with Exception
handling :-)
```

Apprentice level: Pattern matching

Apprentice level: Pattern matching

```
  val foo = request.param("foo") match
  {
  case Some(foo) => foo
  case None => "Default foo"
}
```

Apprentice level: Pattern matching

```
  val foo = request.param("foo") match
  {
  case Some(foo) => foo
  case None => "Default foo"
}
```

Sometimes useful, but...

Apprentice level: Pattern matching

```
  val foo = request.param("foo") match
  {
  case Some(foo) => foo
  case None => "Default foo"
}
```

Sometimes useful, but...

at some point a Jedi you must become

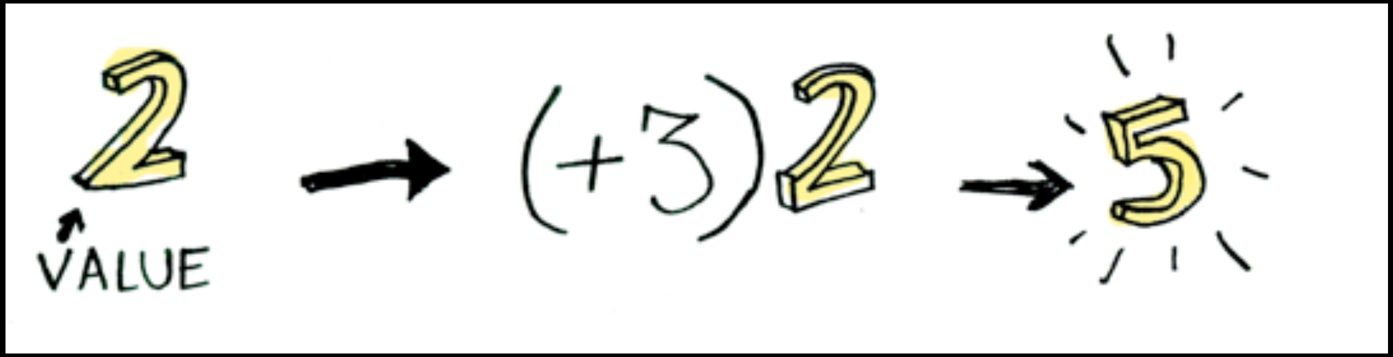
What we really want is

What we really want is

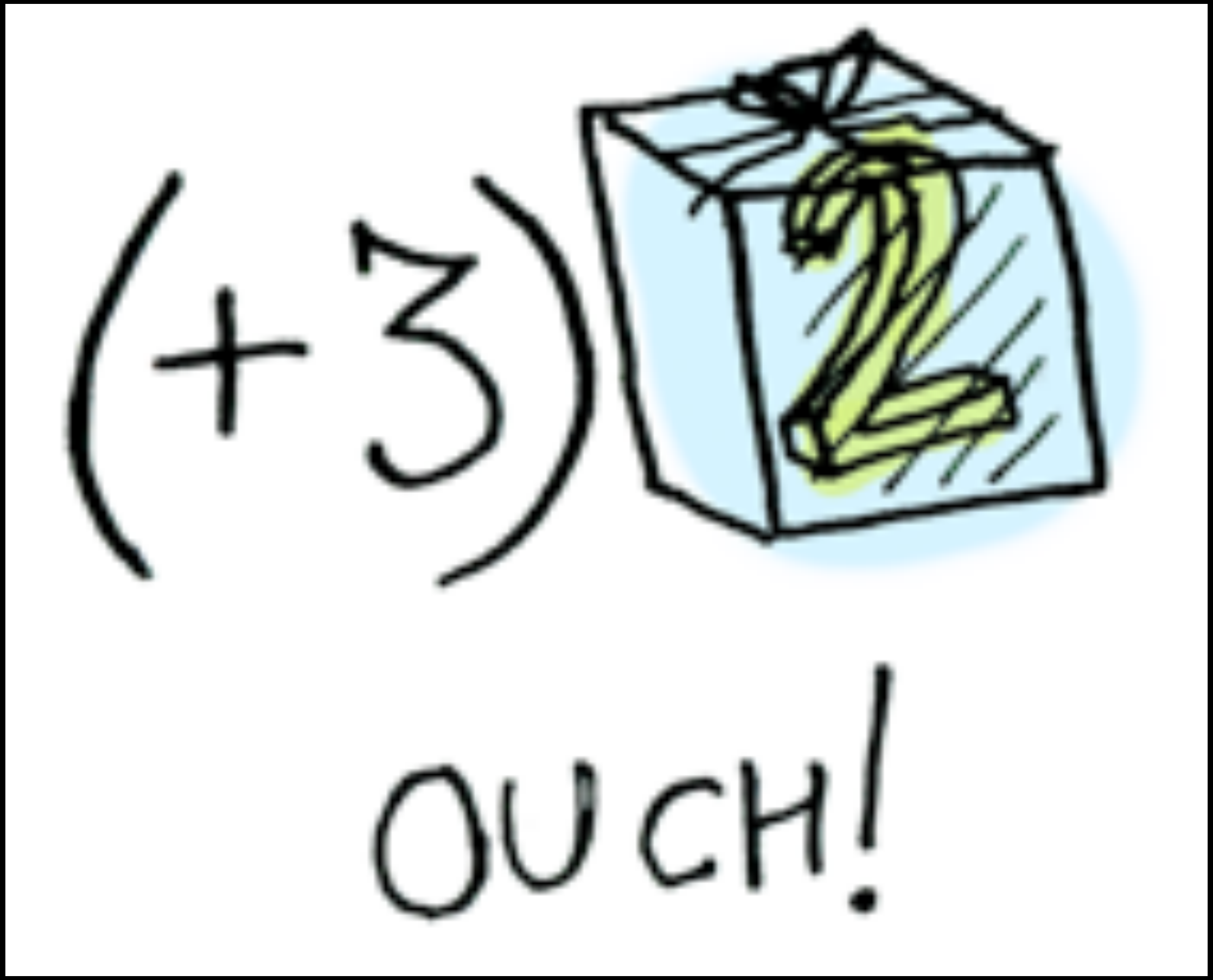
... to do stuff with our values

What we really want is

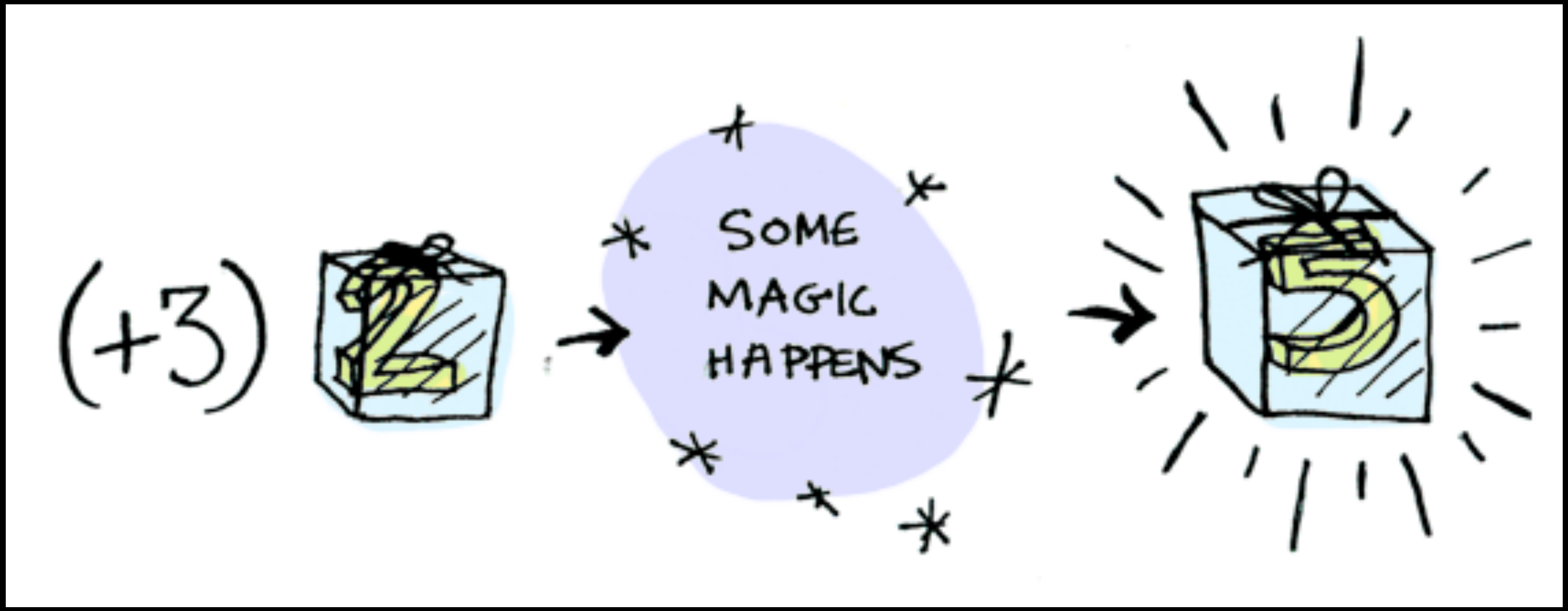
... to do stuff with our values



But...



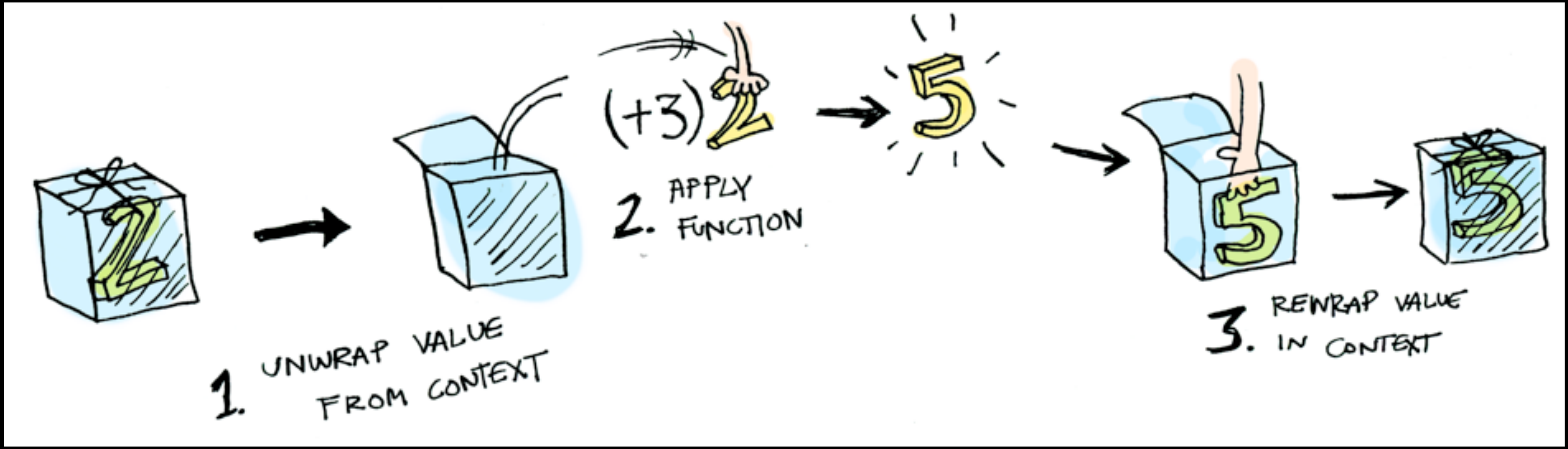
We want...?



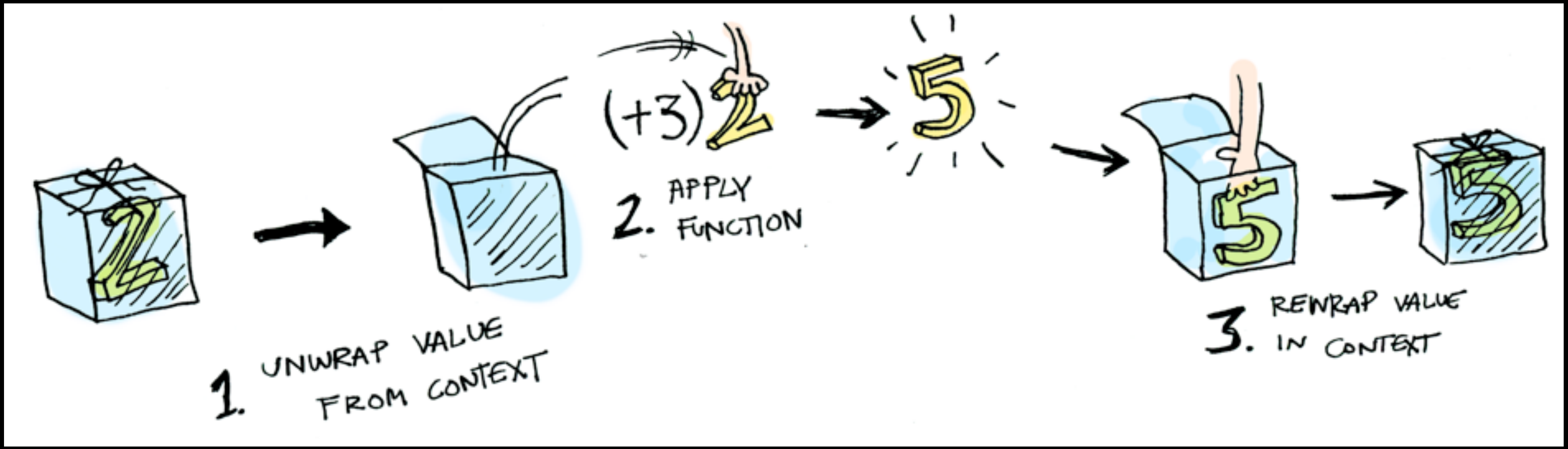
Padawan level: functional

- Treat Option as a (very small) collection
- “Biased” towards Some
- map, flatMap etc.
- and compose to your desire when the option contains a value

map

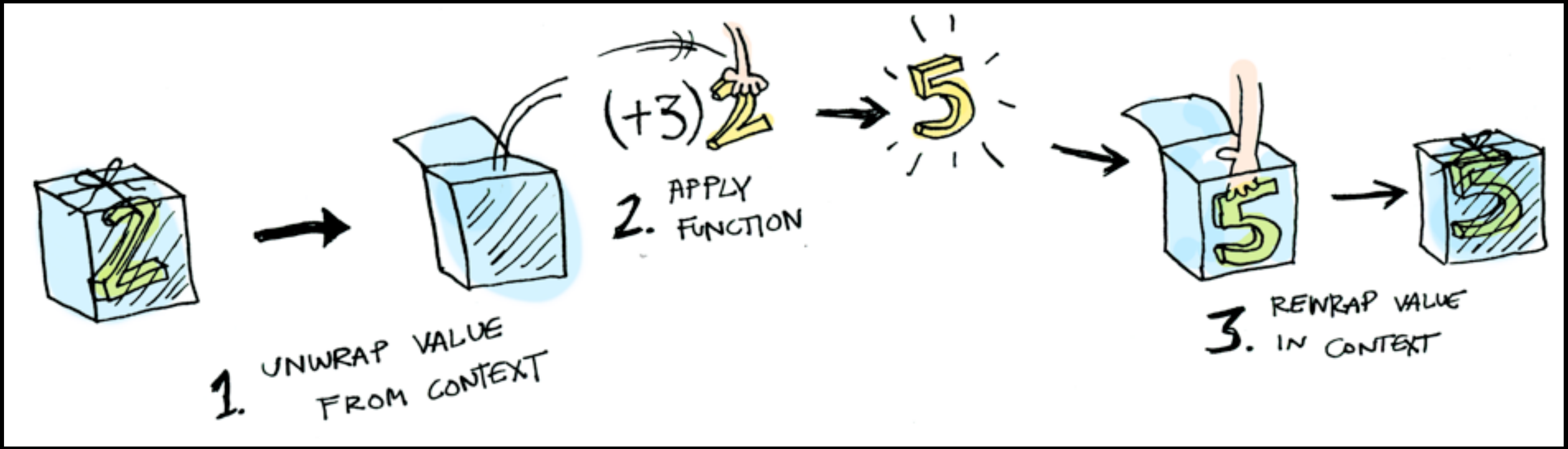

(+3)
2
5
1. UNWRAP VALUE FROM CONTEXT
2. APPLY FUNCTION
3. REWRAP VALUE IN CONTEXT

map



```
val three = Some(3)
    > three : Option[Int] = Some(3)
```

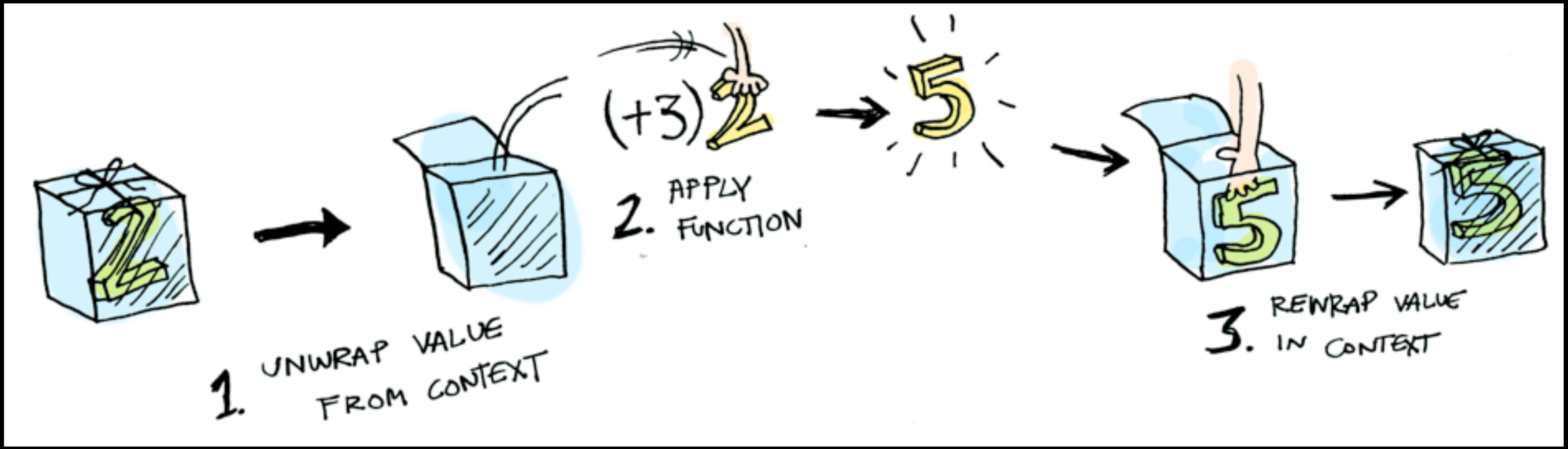
map



```
val three = Some(3)
     > three : Option[Int] = Some(3)

val res = three.map(_ + 3)
```

map



```
val three = Some(3)
     > three : Option[Int] = Some(3)

val res = three.map(_ + 3)


     > res: Option[Int] = Some(6)
```

map

```
option.map(foo(_))
```

equivalent to:

```
option match {
  case None => None
  case Some(x) => Some(foo(x))
}
```

Examples

Examples

```
def sqr(i:Int) = {i*i}
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
        //> res4: Option[Int] = Some(9)
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
        //> res4: Option[Int] = Some(9)
three.map(sqr(_))
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
        //> res4: Option[Int] = Some(9)
three.map(sqr(_))
        //> res5: Option[Int] = Some(9)
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
       //> res4: Option[Int] = Some(9)
three.map(sqr(_))
       //> res5: Option[Int] = Some(9)
three.map(sqr)
```

Examples

```
def sqr(i:Int) = {i*i}
val three = Option(3)

three.map(i => sqr(i))
       //> res4: Option[Int] = Some(9)
three.map(sqr(_))
       //> res5: Option[Int] = Some(9)
three.map(sqr)
       //> res6: Option[Int] = Some(9)
```

flatMap

```
option.flatMap(foo(_))

is equivalent to:

option match {
  case None => None
  case Some(x) => foo(x)
}
```

```
three.flatMap(x => Some(x.toString))
     Option[java.lang.String] = Some(3)

nah.flatMap(x => Some(x.toString))
     Option[java.lang.String] = None
```

Side effects: foreach

```
option.foreach(foo(_))


is equivalent to:

option match {
  case None => {}
  case Some(x) => foo(x)
}
```

```
three.foreach(println(_))
```

```scala
val userOpt = UserDao.findById(userId)

userOpt.foreach(user => println(user.name))


or, even shorter:

userOpt.foreach(println)
```

Working with lists

```
val o1 = Option(1)    //> o1  : Option[Int] = Some(1)
val o2 = Option(2)      //> o2  : Option[Int] = Some(2)
val o3 = Option(3)      //> o3  : Option[Int] = Some(3)
val l = List(o1, nope, o2, nah, o3)
         //> l  : List[Option[Int]]
             = List(Some(1), None, Some(2), None, Some(3))

l.map(_.map(sqr))
     //> res8: List[Option[Int]]
             = List(Some(1), None, Some(4), None, Some(9))

l.flatMap(_.map(sqr))
     //> res9: List[Int] = List(1, 4, 9)
```

Jedi level:
for comprehesions

```
val ageOpt = for {
  user <- UserDao.findById(userId)
  age <- user.ageOpt
} yield age
```

Jedi mind tricks

```
//we have a ‘User’ with mandatory name, but optional age

case class User(val name:String , val age:Option[Int])

def prettyPrint(user: User) =
  List(Option(user.name), user.age).flatten.mkString(", ")

val foo = User("Foo", Some(42))
val bar = User("Bar", None)

prettyPrint(foo)  //prints "Foo, 42"
prettyPrint(bar)  //prints "Bar"
```

```
val userOpt =
    UserDao.findById(userId) OrElse Some(UserDao.create)
```

```
or:
```

```
val user =
  UserDao.findById(userId) getOrElse UserDao.create
```

other option options

```scala
def filter(p: A => Boolean): Option[A]

def exists(p: A => Boolean): Boolean
```

fold
collect
iterator
toList

Resources

References, Thanks, Resources and further reading

- Attributions:
  - Thanks to Adit Bhargava for a great blogpost on monads in Haskel and for letting me use his cartoon drawings: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
  - For broadening my mind on higher-order use of Options: http://blog.tmorris.net/posts/scalaoption-cheat-sheet/
- Further reading
  - http://marakana.com/static/courseware/scala/presentation/comprehending-monads.html
  - http://blog.xebia.com/2011/06/02/scala-options-the-slick-way/