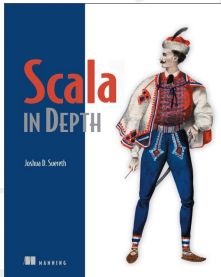
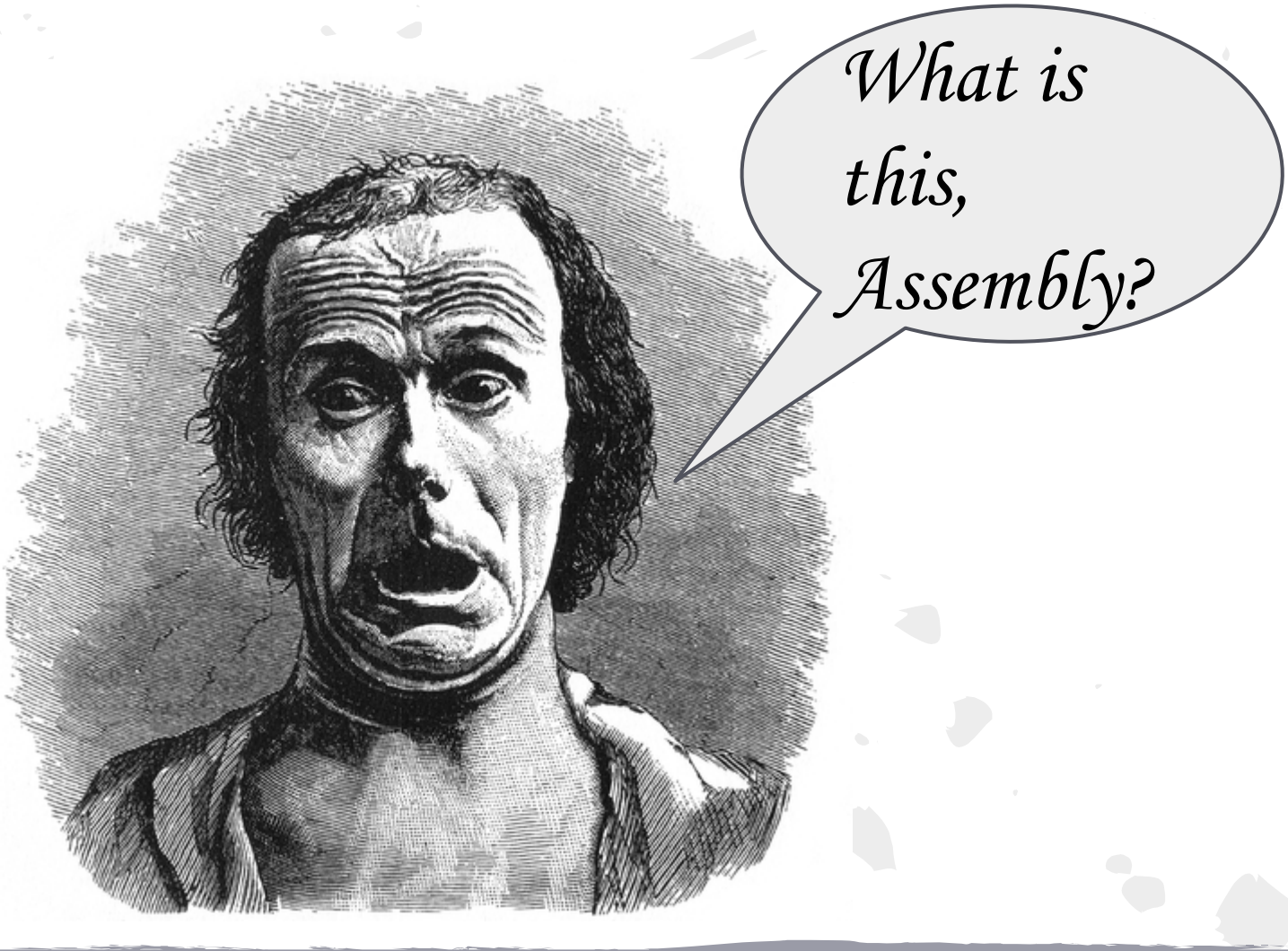
Coding in Style

How to wield Scala in the trenches



Agenda

- Expressions
- Types
- Functions
- Pattern Matching



Scala is all about expressions

Using Expressions

Not statements

Do this. Do that. Do this....

```
def isAwesome(data: Data): Boolean = {
  if(data.awesome) return true
  if(data.lame) return false
  return data.contains("Tasty Sandwich")
}
```

```
def isAwesome(data: Data): Boolean =
  if(data.awesome) true
  else if(data.lame) false
  else (data contains "Tasty Sandwich")
}
```

```
def isAwesome(data: Data): Boolean =
    (data.awesome ||
    (!data.lame)  ||
    (data contains "Tasty Sandwich"))
```

Embrace Operator Notation

```
def privs(user: Option[User]): Privileges =
  user.map(findUserPrivileges).getOrElse(
    NoPrivs)
```

```
def privs(user: Option[User]): Privileges =
   (user map findUserPrivileges
    getOrElse NoPrivs)
```

Avoid

- Postfix operator notation
- Prefix operator notation
- Lack of parenthesis

Let the language do the work

SUDO Make me a variable

```
def slurp(file: File): String = {
  var input: InputStream = null
  var output: OutputStream = null
  var read = 0
  try {
    input = new FileInputStream(file)
    output = new StringOutputStream()
    val buf = new Array[Byte](BUF_SIZE)
    read = input.read(buf)
    while(read > 0) {
       output.write(buf, 0, read)
       read = input.read(buf)
    }
  } finally {
    if(input != null) input.close()
    if(output != null) output.close()
 }
 if(output != null) return output.toString
 return null
}
```

```
def slurp(file: File): String = {
  var input: InputStream = null
  var output: OutputStream = null
  var read = 0
  try {
    input = new FileInputStream(file)
    output = new StringOutputStream()
    val buf = new Array[Byte](BUF_SIZE)
    read = input.read(buf)
    while(read > 0) {
       output.write(buf, 0, read)
       read = input.read(buf)
    }
  } finally {
    if(input != null) input.close()
    if(output != null) output.close()
 }
 if(output != null) return output.toString
 return null
}
```

Create a variable to store the result of the read function

Typesafe

```
def slurp(file: File): String = {
  var input: InputStream = null
  var output: OutputStream = null
  var read = 0
  try {
    input = new FileInputStream(file)
    output = new StringOutputStream()
    val buf = new Array[Byte](BUF_SIZE)
    read = input.read(buf)
    while(read > 0) {
      output.write(buf, 0, read)
      read = input.read(buf)
    }
  } finally {
    if(input != null) input.close()
    if(output != null) output.close()
  }
  if(output != null) return output.toString
  return null
}
```

Store the value

```
def slurp(file: File): String = {
  var input: InputStream = null
  var output: OutputStream = null
  var read = 0
  try {
    input = new FileInputStream(file)
    output = new StringOutputStream()
    val buf = new Array[Byte](BUF_SIZE)
    read = input.read(buf)
    while(read > 0) {
      output.write(buf, 0, read)
      read = input.read(buf)
    }
  } finally {
    if(input != null) input.close()
    if(output != null) output.close()
  }
  if(output != null) return output.toString
  return null
}
```

Read the value right after storing

Typesafe

```
def slurp(in: File): String = {
  val in = new FileInputStream(in)
  val out = new StringWriter()
  val buf = new Array[Byte](BUF_SIZE)
  def read(): Unit =
    (in read buf) match {
      case 0 => ()
      case n =>
        out.write(buf, 0, n)
        read()
    }
  try read()
  finally in.close()
  out.toString
}
```

```
def slurp(in: File): String = {
  val in = new FileInputStream(in)
  val out = new StringWriter()
  val buf = new Array[Byte](BUF_SIZE)
  def read(): Unit =
    (in read buf) match {
      case 0 => ()
      case n =>
        out.write(buf, 0, n)
        read()
    }
  try read()
  finally in.close()
  out.toString
}
```

Use the value immediately

Typesafe

Tail-recursioN!

```
@tailrec
def read(): Unit =
  (in read buf) match {
    case 0 => ()
    case n =>
      out.write(buf, 0, n)
      read()
  }
```

Tail-recursioN!

Pattern match on the value

```
@tailrec
def read(): Unit =
  (in read buf) match {
    case 0 => ()
    case n =>
      out.write(buf, 0, n)
      read()
  }
```

Typesafe

Tail-recursioN!

```
@tailrec
def read(): Unit =
  (in read buf) match {
    case 0 => ()
    case n =>
      out.write(buf, 0, n)
      read()
  }
```

Base Case

Typesafe

Tail-recursioN!

```
@tailrec
def read(): Unit =
  (in read buf) match {
    case 0 => ()
    case n =>
      out.write(buf, 0, n)
      read()
  }
```

Recursive Case

Typesafe

Tail-recursioN!

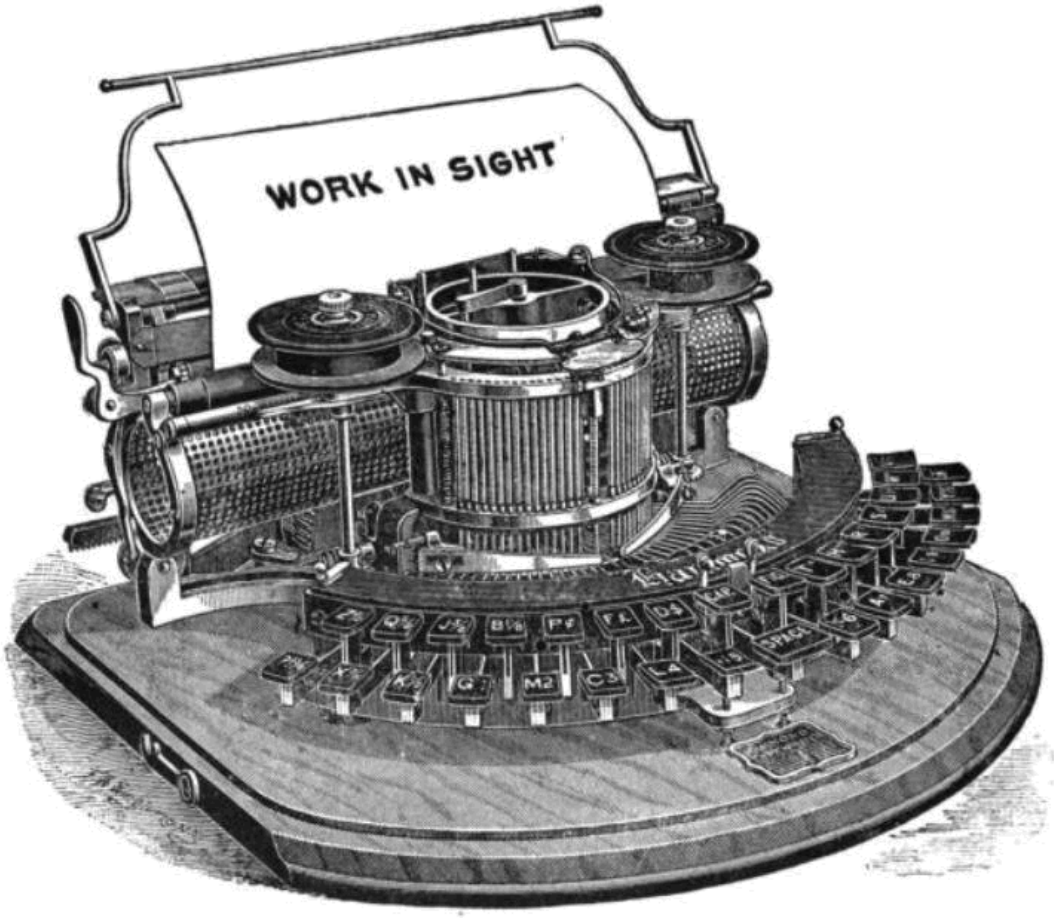
```
@tailrec
def read(): Unit =
  (in read buf) match {
    case 0 => ()
    case n =>
      out.write(buf, 0, n)
      read()
  }
```

Capture the value in a variable

Typesafe

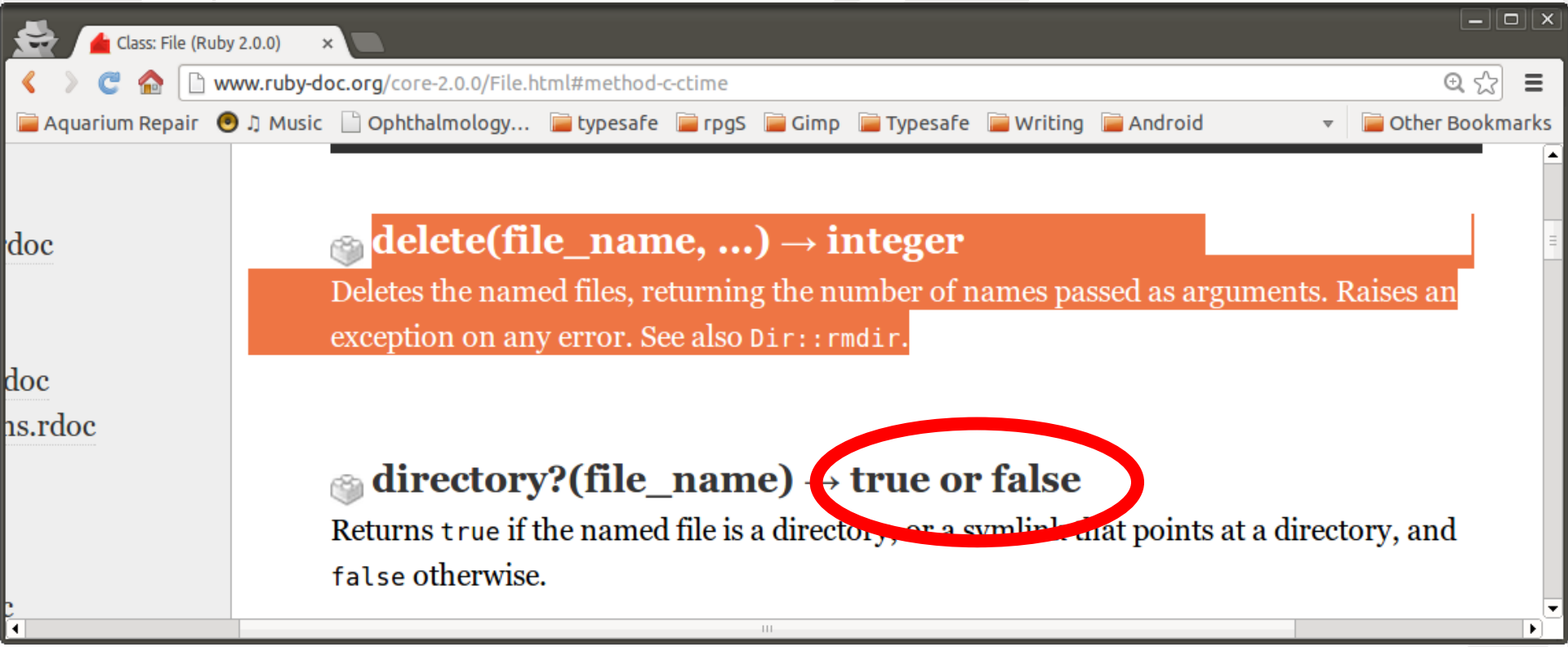
Change your thinking

- Everything returns something
- What should I be returning?



*When you've got **TYPES**, lean on em*

Ruby API docs



Types are Documentation

They should provide value.

ZOMG TYPE INFERENCE!!!

```
def mainMethod(cl: Class[_])=
  try {
    val method =
      clazz.getMethod("main",
                      classOf[Array[String]])
    if(isStatic(method.getModifiers)){
      method
    } else None
  } catch {
    case n: NoSuchMethodException => None
  }
```

Mental type gymnastics

```
def mainMethod(cl: Class[_])=
  try {
    val method =
      clazz.getMethod("main",
                      classOf[Array[String]])
    if(isStatic(method.getModifiers)){
     method
    } else None
  } catch {
    case n: NoSuchMethodException => None
  }
```

Code readability

- Types are great documentation for interfaces
- Your coworkers will love you if your types are well named, and annotated for interfaces they need.

Unexpected type is returned

```
def mainMethod(cl: Class[_]): AnyRef =
  try {
    val method =
      clazz.getMethod("main",
                      classOf[Array[String]])
    if(isStatic(method.getModifiers)){
     method
    } else None
  } catch {
    case n: NoSuchMethodException => None
  }
```

Annotate your return types

It's documentation

Annotate your types!

```
def mainMethod(cl: Class[_]): Option[Method]
=
  try {
    val method =
      clazz.getMethod("main",
                      classOf[Array[String]])
    if(isStatic(method.getModifiers)){
      method
    } else None
  } catch {
    case n: NoSuchMethodException => None
  }
```

Type Error!
Expected: `Option[Method]`
Found: `Method`

Annotate your types!

```
def mainMethod(cl: Class[_]): Option[Method]
=
  try {
    val method =
      clazz.getMethod("main",
                      classOf[Array[String]])
    if(isStatic(method.getModifiers)){
      Some(method)
    } else None
  } catch {
    case n: NoSuchMethodException => None
  }
```

Why annotate?

- Documentation
- Better error messages
- Don’t leak abstractions

Use meaningful types

On the overuse of
String, Boolean, tuples and null

A bad example of types

```
/**
 * @return
 *      null if data.length < 1
 *      The mean, median and stddev
 *      otherwise.
 */
def statistics(data: Seq[Double]):
(Double, Double, Double) = ….
```

A better example of types

```
case class Statistics(
  mean: Double,
  median: Double,
  standard_deviation: Double)
/**
 * @return
 *      null if data.length < 1
 *      The statistics otherwise
 */
def statistics(data: Seq[Double]): Statistics
=
```

An even better example

```
case class Statistics(
  mean: Double,
  median: Double,
  standard_deviation: Double)
def statistics(
  data: Seq[Double]): Option[Statistics] =
  ….
```

Better Types

- ***Case classes*** over raw types
- `Option` instead of null to represent ***absence*** of a return.
- `Option` is ***not*** an error reporting mechanism

Failure Handling

- Try
  - Two states
    - Success(result)
    - Failure(exception)
  - Automatically captures exceptions in Failure State
- Validation
  - Two States
    - Success(Result)
    - Failure(Errors)
  - Can aggregate failures
  - Failures must be explicitly created

Try Example

```
val first: Try[String] =
  Try(Console.readLine(
      "Enter a number:"))
val second: Try[String] =
  Try(Console.readLine(
      "Enter another number:"))
val sum: Try[Int] =
  for {
     f <- first
     s <- second
  } yield f.toInt + s.toInt
```

Try Example

```
sum match {
  case Success(sum) =>
    println(s"Sum is ${sum}")
  case Failure(ex) =>
    println("Not a valid number")
}
```

Try

- *Automatically* swallow non-fatal errors
- *Forces* consumer to handle failure case
- Eager evaluation (happens right away)
- Failure is ***always*** a single exception instance
  - Multiple failures have to be encoded in an exception instance.

scalaz.Validation

```
sealed trait Validation[Error, Result]

case class Success[Error,Result](
  result: Result)
  extends Validation[Error,Result]

case class Failure[Error,Result](
  error: Error)
  extends Validation[Error, Result]
```

Validation

- Errors can be any type
  - Could be a List[Error]
  - Could be an Exception
- There's an additional type parameter to track: `Error` vs. just `Result` for `Try`
- If we use a "joinable" error type, we can join together errors rather than picking the first one

Joining errors

```
val user = getUser(form)
val pw = getPassword(form)


(user |@| pw) apply { (u, p) =>
   getCredentials(u,p)
}
```

Design isn't just about nouns. Functions are people too

Functional Design

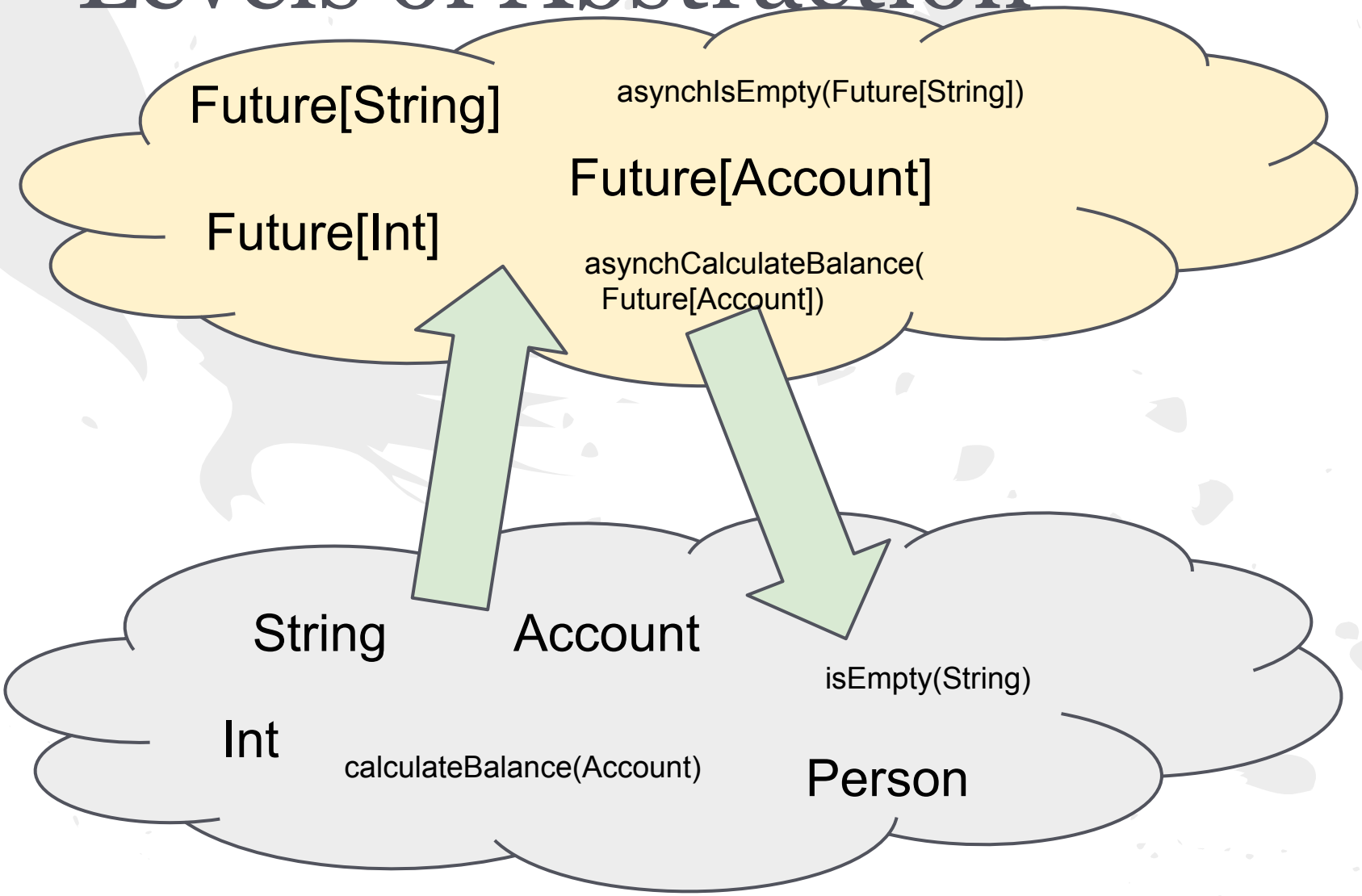
- Reusable base of functions
  - Testable in isolation

```
def sum(a: Int, b: Int): Int
```

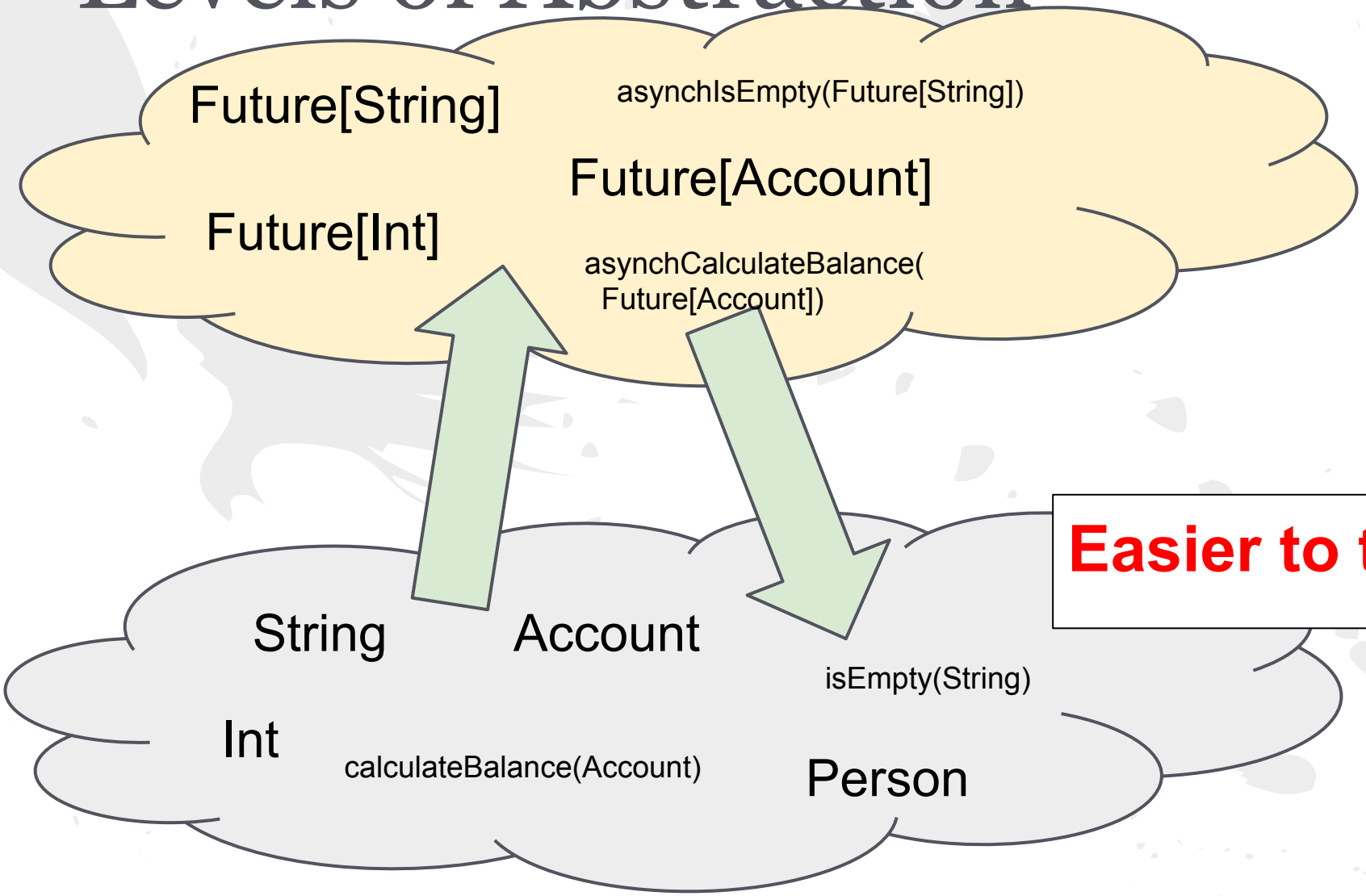
- "Lift" more generic functions into more specific ones.
  - Harder to test directly.

```
def asynchSum(a: Future[Int],
              b: Future[Int]): Future[Int] =
  for {
     aResult <- a
     bResult <- b
  } yield sum(aResult, bResult)
```

Levels of Abstraction

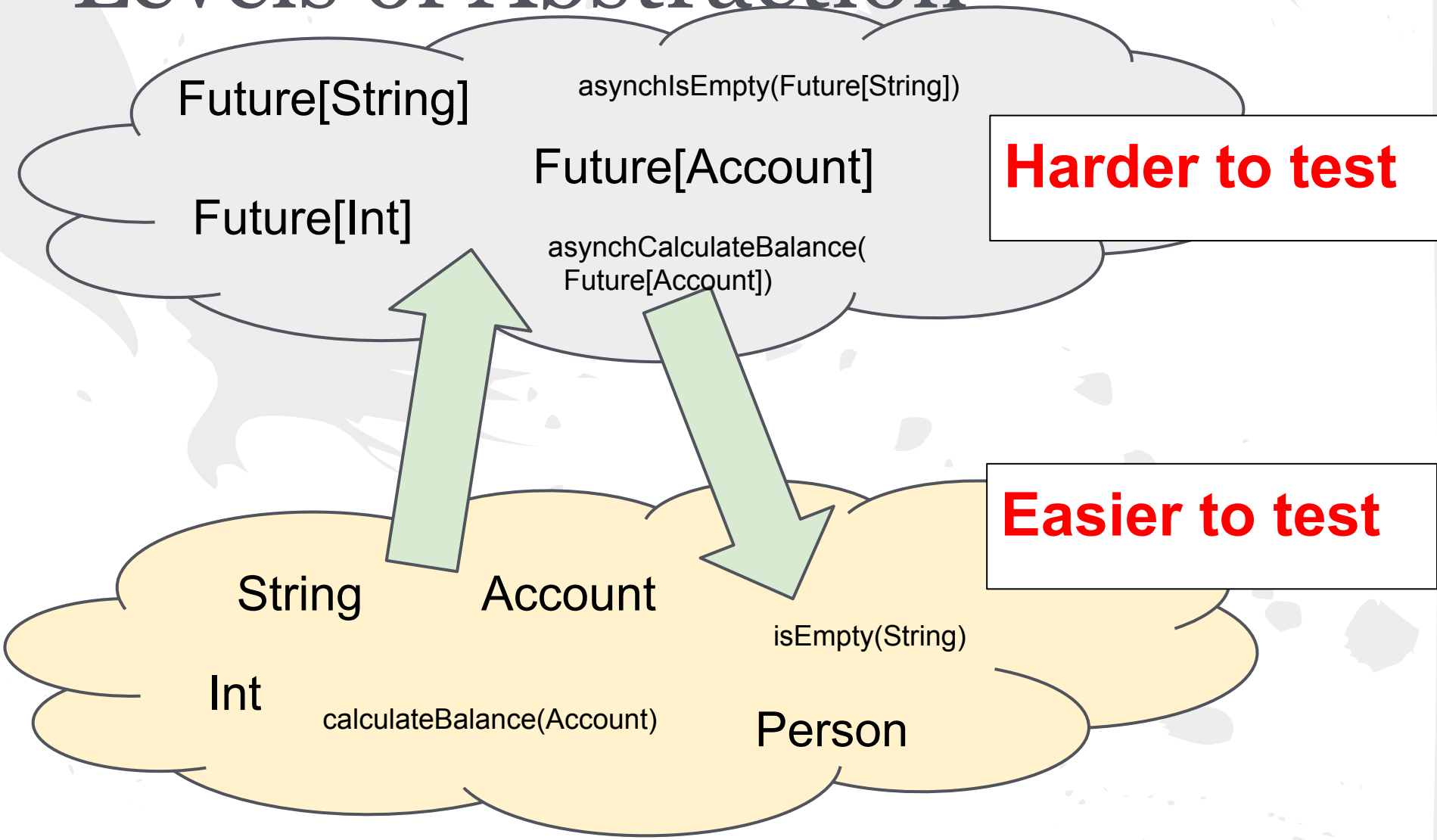


Levels of Abstraction



Easier to test

Levels of Abstraction

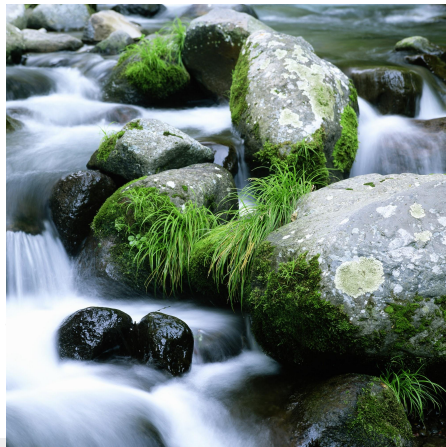


Functional Design

- “Lifting” and “Digging” between abstraction levels
  - Test the lowest level function
  - Test the ability to “lift” into an abstraction
    - There are laws that, if followed, guarantee safe transitions

Flowing Data

Keeping external concerns external



Asycnhronize

```
def calculateResult(): Future[Result] = {
  for {
    user <- Future(getUser())
    ads  <- asyncGetAds(user, getCity)
  } yield Html(user, ads)
```

Asycnhronize

```
def calculateResult(): Future[Result] = {
  for {
    user <- Future(getUser())
    ads  <- asyncGetAds(user, getCity)
  } yield Html(user, ads)
```

Lifting

Asycnhronize

```
def calculateResult(): Future[Result] = {
  for {
    user <- Future(getUser())
    ads  <- asyncGetAds(user, getCity)
  } yield Html(user, ads)
```

Digging

For-Expressions

- Scala's mechanism of digging into functional abstractions so you can work at the lower level
- Are sequencing operations
  - Does not do operations in parallel.
  - First extraction must happen before the second
- Are wrappers over generic methods:
  - map (andThen)
  - flatMap (andThenCompose)
  - filterWith (no java equivalent)

Conditional logic via patterns

Pattern matching

- Scala's secret power
- Selectively execute behavior based on data
- Selectively extract values from data

Example

Let's take a list of people and find everyone in Pittsburgh

```
case class Person(name: String,
                  residence: Seq[Residence])

case class Residence(city: String,
                     country: String)
```

Example

Let's take a list of people and find everyone in Pittsburgh

```
val people: Seq[Person] = …

people collect {
  case ??? => ???
}
```

First, let's extract cities

```
object LivesIn {
  def unapply(p: Person): Option[Seq[String]] =
      Some(
        for(r <- p.residences)
        yield r.city
      )
}
```

Example

Let's take a list of people and find everyone in Pittsburgh

```
val people: Seq[Person] = …

people collect {
  case LivesIn(cities) if ??? => ???
}
```

Next, let's extract a particular city

```
class StringSeqContains(value: String) {
  def unapply(in: Seq[String]): Boolean =
      in contains value
}
```

Example

Let's take a list of people and find everyone in Pittsburgh

```
val people: Seq[Person] = …
val Pittsburgh =
    new StringSeqContains("Pittsburgh")

people collect {
  case LivesIn(Pittsburgh) => ???
}
```

Example

Let's take a list of people and find everyone in Pittsburgh

```
val people: Seq[Person] = …
val Pittsburgh =
    new StringSeqContains("Pittsburgh")

people collect {
  case person @ LivesIn(Pittsburgh) =>
    person
}
```

Pattern matching

- Can be used for conditional logic
  - `match` statement
  - `catch` statement
  - `partial` functions
- Can be used to extract into variables
  - `val LivesIn(cities) = person`
- Don’t forget `Seq#collect`

RECAP

- Think in expressions
- Leverage the types
- Composition of functions
  - Generic methods easier to test
  - Lift and Dig into abstractions as needed
- Rip into data with Pattern matching.

Questions?