Functional Error Handling

Georgia Tech

What's right with exceptions?

Exceptions provide

- a way to consolidate error handling code and separate it from main logic, and
- an alternative to APIs that require callers of functions to know error codes, sentinal values, or calling protocols.

We can preserve both of these advantages while avoiding the disadvantages of exceptions.

Georgia Tech

What's wrong with exceptions?

Exceptions

- break referential transparency,
- are not type-safe, and
- functions that throw excpetions are *partial*.

Also, exception syntax is a pain.

Georgia Tech

Exceptions break referential transparency.

```scala
def failingFn(i: Int): Int = {
  val y: Int = throw new Exception("fail!")
  try {
    val x = 42 + 5
    x + y
  } catch {
    case e: Exception => 43
  }
}
```

If `y` were referentially transparent, then we should be able to substitute the value it references:

```scala
def failingFn2(i: Int): Int = {
  try {
    val x = 42 + 5
    x + ((throw new Exception("fail!")): Int)
  } catch {
    case e: Exception => 43
  }
}
```

But `failingFn2` returns a different result for the same input.

Type-safety and Partiality

```
1  def mean(xs: Seq[Double]): Double =
2    if (xs.isEmpty)
3      throw new ArithmeticException("mean of empty list undefined")
4    else
5      xs.sum / xs.length
```

`mean(Seq(1,2,3))` returns a value, but `mean(Seq())` throws an exception

- The type of the function, `Seq[Double] => Double`, does not convey the fact that an exception is thrown in some cases.
- `mean` is not defined for all values of `Seq[Double]`.

In practice, partiality is common, so we need a way to deal with it.

Georgia Tech

Functional Error Handling in the Scala Standard Library

The Scala standard library defines three useful algebraic data types for dealing with errors:

- `Option`, which represents a value that may be absent,
- `Either`, which represents two mutually-exclusive alternatives, and
- `Try`, which represents success and failure

Note: Chapter 4 of Functional Programming in Scala defines its own parallel versions of `Option` and `Either`, but we'll use the standard library versions. For a deeper understanding do the exercises in the book.

Georgia Tech

The `Option` Type

We've seen `Option` before:

```
1  sealed abstract class Option[+A]
2  final case class Some[+A](value: A) extends Option[A]
3  case object None extends Option[Nothing]
```

Using `Option`, `mean` becomes

```
1  def mean(xs: Seq[Double]): Option[Double] =
2    if (xs.isEmpty) None
3    else Some(xs.sum / xs.length)
```

Georgia Tech

`Option`'s Definition

`Option` defines many methods that mirror methods on `Traversable`s.

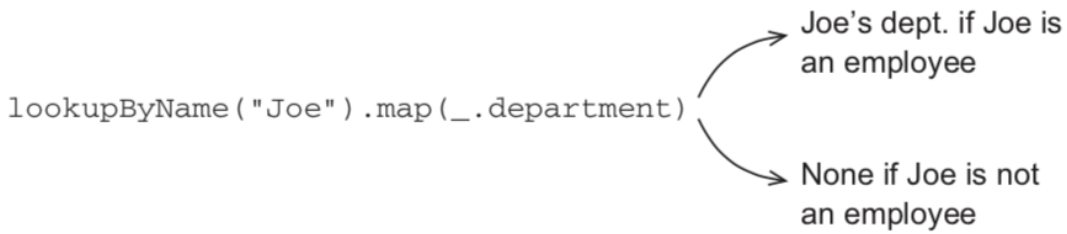
```
1  sealed abstract class Option[+A] {
2    def isEmpty: Boolean
3    def get: A
4
5    final def getOrElse[B >: A](default: => B): B =
6      if (isEmpty) default else this.get
7
8    final def map[B](f: A => B): Option[B] =
9      if (isEmpty) None else Some(f(this.get))
10
11   final def flatMap[B](f: A => Option[B]): Option[B] =
12     if (isEmpty) None else f(this.get)
13
14   final def filter(p: A => Boolean): Option[A] =
15     if (isEmpty || p(this.get)) this else None
16 }
```

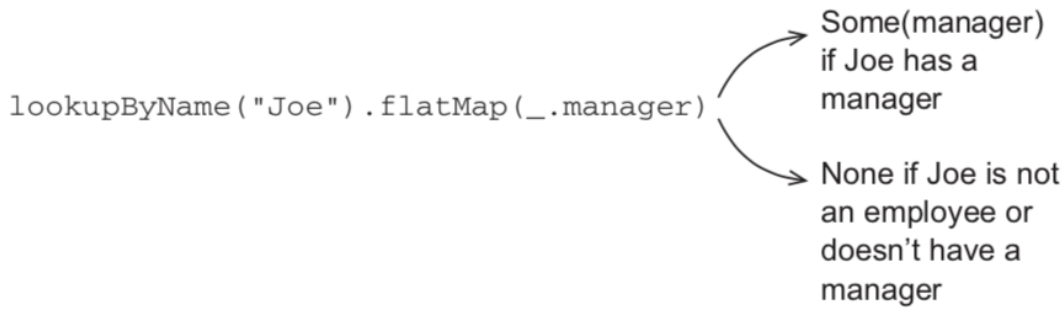
The key consequence is that you can treat `Option` as a collection, leading to Scala's idioms for handling optional values.

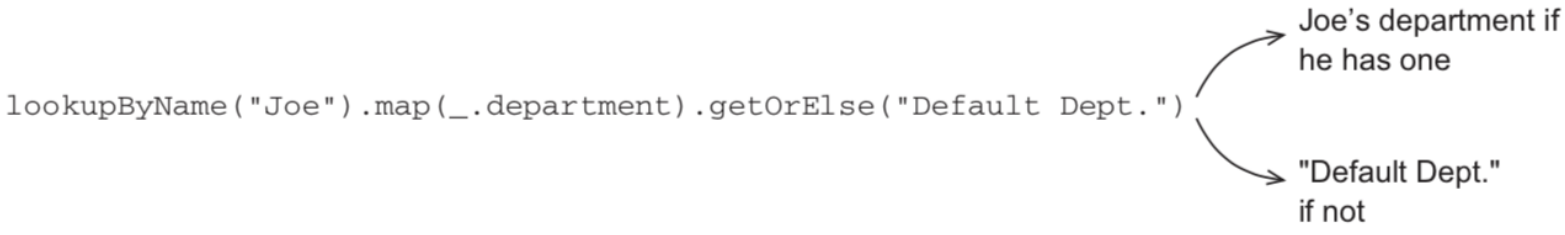
Georgia Tech

`Option` Examples

```
1  case class Employee(name: String, department: String)
2
3  def lookupByName(name: String): Option[Employee] = // ...
4
5  val joeDepartment: Option[String] =
       lookupByName("Joe").map(_.department)
```







`Option` Idioms

```
1  case class Employee(name: String, department: String)
2
3  def lookupByName(name: String): Option[Employee] = // ...
4
5  val joeDepartment: Option[String] =
       lookupByName("Joe").map(_.department)
```

```
1  val dept: String =
2    lookupByName("Joe").
3    map(_.dept).
4    filter(_ != "Accounting").
5    getOrElse("Default Dept")
```

The `getOrElse` at the end returns `"Default Dept"` if Joe doesn't have a department, or if Joe's department is not `"Accounting"`.

Georgia Tech

Dealing with Exception-Oriented APIs

```
1  scala> import scala.util.Try
2  import scala.util.Try
3
4  scala> Try { "foo".toInt }
5  res1: scala.util.Try[Int] = Failure(java.lang.NumberFormatException:
       For input string: "foo")
6
7  scala> Try { "1".toInt }
8  res2: scala.util.Try[Int] = Success(1)
```

Georgia Tech

EitherS

Return error message on failure:

```
1  def mean(xs: IndexedSeq[Double]): Either[String, Double] =
2    if (xs.isEmpty)
3      Left("mean of empty list!")
4    else
5      Right(xs.sum / xs.length)
```

Return the exception itself on failure:

```
1  def safeDiv(x: Int, y: Int): Either[Exception, Int] =
2    try Right(x / y)
3    catch { case e: Exception => Left(e) }
```

Georgia Tech

Closing Thoughts

Rule of thumb: only throw exceptions exceptions in cases where the program could not recover from the exception by catching it.

Georgia Tech