Maybe you should think about your Options

A one-class play in two acts.

About Me

- Kris Nuttycombe
  - Newly independent consultant looking for clients who are interested in adopting Scala and/or functional programming techniques
  - Been using Scala in production for 2+ years
  - Committer to the Lift web framework
  - Committer to Apache Commons
  - kris@hylotech.com, @nuttycom on Twitter & elsewhere

Act One:
Deriving the Option

- What is the most common RuntimeException?

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Tony Hoare

"Option and Maybe are both derived from mathematics….
whereas null is derived from Satan's bottom."

Neil Bartlett

- Null is a master impersonator; it is considered by the compiler to have a type that is subtype of every reference type.

- It rips a jagged hole in our type system. NPEs are the cuts you get from the edge of that hole.

- Now, such a type can be useful - this is known as a bottom type: ⊥ (Scala calls this Nothing)

- It is used to represent the result of a function that does not return normally.

- But the bottom type should have no instances! Null is the instance of the bottom type - its type is a lie!

- It doesn't have to be this way.
- find . -name *.java | xargs grep null
- Then, remove every single one.*

*some restrictions may apply

- Never pass null as an argument to a method.

- Never return null from a method.

- Replacements for null?
  - The Null Object Pattern (Collections.emptySet() etc.) - good, but often clumsy to use for non-collection data types.
  - @NotNull in Java 7 - also good, but it misses a very common use case - data that might *intentionally* be absent!

- Use Option<A> instead.

- Option[A] (Scala), Maybe a (Haskell doesn't have null at all!)

- Option<A> is an *algebraic data type* whose members are Some<A> and None<A>

- Option<A> is a container that is either empty, or contains a single value of type A.

Live Coding

```
abstract class Option<A> implements Iterable<A> {
  Iterator<A> iterator();
  A getOrElse(A a);
}
```

- I lied. The most important thing about Option<A> is not actually null safety. That's just a bonus. And iterating to get the value out sucks.

- The compositional properties of Option<A> are much more interesting.

- Let's add some *higher-order functions*

```
interface F<A, B> {
  B apply(A a);
}

abstract class Option<A> implements Iterable<A> {
  ...
  void foreach(F<A,Void> f);
  Option<A> filter(F<A, Boolean> f);
}
```

```scala
// a generalized definition of what a functor is
trait Functor[M[_]] {
  def map[A,B](m: M[A], f: A => B): M[B]
}

// might look like this in Java
interface Functor<A> {
  <B> Functor<B> map(F<A, B> f);
}

abstract class Option<A> implements Functor<A> {
  //...
  <B> Option<B> map(F<A, B> f);
}
```

```
trait Monad[M[_]] extends Functor[M] {
  def pure[A](a: A): M[A]
  def flatMap[A, B](m: M[A], f: A => M[B]): M[B]
}

// this doesn't quite work:
interface Monad<A> extends Functor<A> {
  <B> Monad<B> flatMap(F<A, Monad<B>> f);
}

abstract class Option<A> implements Iterable<A> {
  //...
  <B> Option<B> flatMap(F<A, Option<B>> f);
}
```

```
interface F<A, B> {
  B apply(A a);
}

abstract class Option<A> implements Iterable<A> {
  Iterator<A> iterator();
  A getOrElse(A a);

  void foreach(F<A,Void> f);
  Option<A> filter(F<A, Boolean> f);
  <B> Option<B> map(F<A, B> f);
  <B> Option<B> flatMap(F<A, Option<B>> f);
}
```

Act Two:
The Catamorphism At The Heart of Everything

- This is just geeky fun.

Rubyists should find this very familiar:

```
module Enumerable
  def inject(initial) {|memo, obj| block}
end
```

```
interface F<A, B> {
  B apply(A a);
}

interface F2<A, B, C> {
  C apply(A a, B b);
}

abstract class Option<A> implements Iterable<A> {
  // the catamorphism for Option
  <B> B fold(B b, F2<A, B, B> f);
}
```

```
abstract class Option<A> implements Iterable<A> {
  //these are isomorphic in the context of Option
  <B> B fold(B b, F2<A, B, B> f);
  <B> B fold(B b, F2<A, B> f);
  <B> F<B, B> fold(F<A, B> f, F<B, B, B> append);
}
```

```
sealed trait Option[A] {
  def fold(some: A => B, none: B): B
}
```

```scala
sealed trait Option[A] {
  // looks a lot like getOrElse, but as a HOF
  def fold(some: A => B, none: B): B
}

object Option {
  def some[A](a: A) = new Option[A] {
    override def fold[B](s: A => B, n: => B): B = s(a)
  }

  def none[A] = new Option[A] {
    override def fold[B](s: A => B, n: => B): B = n
  }
}
```

```
sealed trait Option[A] {
  def fold(some: A => B, none: B): B

  // all of these can be defined in terms of fold
  // and the factory methods
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  def getOrElse[AA >: A](e: => AA): AA
  def filter(p: A => Boolean): Option[A]
  def foreach(f: A => Unit): Unit
  def isEmpty: Boolean
  def iterator: Iterator[A]
}
```

- For more information:
  - http://functionaljava.org/
  - http://www.scala-lang.org/api
  - http://code.google.com/p/scalacheck
  - http://code.google.com/p/specs
  - http://code.google.com/p/scalaz

- Special thanks to Tony Morris for the definition of Option in terms of its catamorphism and to Rúnar Bjarnason for his posts on monoids and monads.

- http://blog.tmorris.net/debut-with-a-catamorphism/

- http://apocalisp.wordpress.com/2010/06/14/on-monoids/