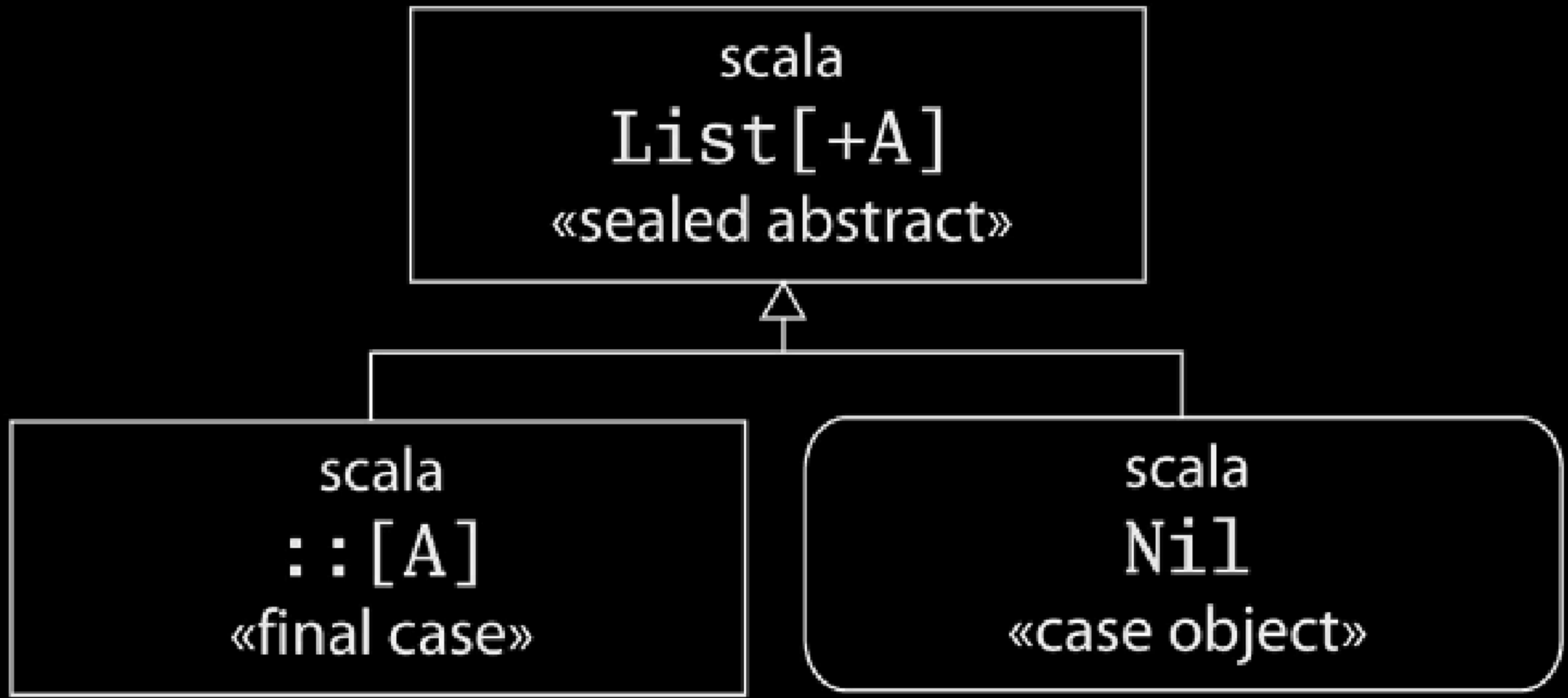
Implementing Lists in Scala

Overview of Content:

- List Hierarchy
- Nil object
- :: class
- :: method
- Accessing Lists
- ListBuffer


scala
List[+A]
«sealed abstract»
scala
::[A]
«final case»
scala
Nil
«case object»

The `Nil` object

It has a straight forward implementation.

Nil can be apart of any list because of co-variance.

`::` class vs `::` method

`scala.::` is a class that has a constructor defined because it is a case class.

```
scala> scala.::(1, Nil)
val res0: scala.collection.immutable.::[Int] = List(1)

scala> new ::(1, Nil)
val res1: scala.collection.immutable.::[Int] = List(1)
```

Like Nil, this also has a straight forward implementation.

`::` is an infix, list construction method.

```
scala> 1 :: Nil
val res0: List[Int] = List(1)

scala> Nil.::(1)
val res1: List[Int] = List(1)
```

How is this implemented?

Why use [B >: A] and not [A]?

Consider the following:

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit

scala> val apples = new Apple :: Nil
apples: List[Apple] = List(Apple@e885c6a)
scala> val fruits = new Orange :: apples
fruits: List[Fruit] = List(Orange@3f51b349, Apple@e885c6a)
```

Lists have a natural recursive structure. This means accessing lists in a recursive manner appears to be logical.

The main problem is that (for non-tail recursive functions) a new stack frame is created for each call.

The solution is to use an imperative approach.

Lists end up using imperative approaches for their methods, such as map.

Another effecient way to construct lists is to use a `ListBuffer`.

ListBuffer constructs an internal list, it does this by modifiying (!) the tail when needed.

Functional on the outside.

Questions?