Value Classes in Scala

Lukas Rytz, Typesafe

Scala

Typesafe

Value Classes in Scala

- Introduced in Scala 2.10.0 (January 2013)
- Release Notes:

> A class may now extend `AnyVal` to make it behave like a struct type (restrictions apply).

Agenda

- Introduction to value classes
- Compiler transformation: an example
- Value classes in Scala's type hierarchy
- Limitations and feature interactions

Main Idea

- Value class instances are *inlined:* represented at runtime as their fields
  - ↣ Space: no object header, no work for GC
  - ↣ Locality: no pointer chasing
  - ↣ Convenient for programmers (normal classes)

Implications of this Representation

- Values (value class instances) have no object identity
  - ↣ No locking on values, identityHashCode, ...
- Fields of values are immutable
  - ↣ Values are passed "by value" (copied)

Some Use Cases

- Extension methods without overhead
- Bit fields: Mode in scalac's type checker, Flags
- Semantic primitives: units of measure
- New numeric types like Complex
  - ↣ FastComplex in *spire*: two floats in one long
  - ↣ (Scala has only single-field value classes)

Agenda

- Introduction to value classes
- Compiler transformation: an example
- Value classes in Scala's type hierarchy
- Limitations and feature interactions

Example: Extension Method

```
object Predef {
  implicit class Ar[A](private val a: A)
  extends AnyVal {
    def -> [B](b: B): Tuple2[A, B] = Tuple2(a, b)
  }
}

scala> "jan" -> "january"
res0: (String, String) = (jan,january)
```

Example: extension method

```
object Predef {
  implicit class Ar[A](private val a: A)
  extends AnyVal {
    def -> [B](b: B): Tuple2[A, B] = Tuple2(a, b)
  }
}
```

```
scala> "jan" -> "january"
res0: (String, String) = (jan,january)
```

- **Parser**
- Type Checker
- Extension Methods
- Erasure

-Xprint:parser

```
implicit class Ar[A] extends AnyVal {
  private val a: A = _
  def <init>(a: A) = super.<init>()
  def ->[B](b: B): Tuple2[A, B] = Tuple2(a, b)
}
```

- Parser
- **Type Checker**
- Extension Methods
- Erasure

-Xprint:typer

```
final class Ar[A] extends AnyVal {
  private[this] val a: A = _
  private def a: A = this.a

  def <init>(a: A): Ar[A] = super.<init>()
  def ->[B](b: B): (A, B) = Tuple2.apply[A, B](a, b)

  override def hashCode(): Int = ...
  override def equals(x$1: Any): Boolean = ...
}

implicit def Ar[A](a: A): Ar[A] = new Ar[A](a)
```

- Parser
- Type Checker
- **Extension Methods**
- Erasure

-Xprint:extmethods

```
final class Ar[A] extends AnyVal {
  // Field, constructor

  def ->[B](b: B): (A, B) =
    Ar.->$extension[B, A](this)(b)

  // Similar for hashCode, equals
}

object Ar extends AnyRef {
  def ->$extension[B, A]($this: Ar[A])(b: B): (A, B) =
    Tuple2.apply[A, B]($this.a, b)

  // similar for hashCode, equals
}

implicit def Ar[A](a: A): Ar[A] = new Ar[A](a)
```

- Parser
- Type Checker
- Extension Methods
- **Erasure**

-Xprint:posterasure

Ar[A] ⟹ Ar

A ⟹ Object

```
final class Ar extends Object {
  private[this] val a: Object = _
  final def a(): Object = this.a
  def <init>(a: Object): Ar = super.<init>()
  def ->(b: Object): Tuple2 = Ar.->$extension(this.a(), b)
}

object Ar extends Object {
  def ->$extension($this: Object, b: Object): Tuple2 =
    new Tuple2($this, b)
}
implicit def Ar(a: Object): Object = a
```

Ar[A] ⟹ A ⟹ Object

$this.a() ⟹ $this

new Ar(a) ⟹ a

Example: extension method

```
object Predef {
  implicit class Ar[A](private val a: A)
  extends AnyVal {
    def -> [B](b: B): Tuple2[A, B] = Tuple2(a, b)
  }
}
```

```
scala> "jan" -> "january"
res0: (String, String) = (jan,january)
```

Client Code

```
class C { def f = "jan" -> "january" }

// After typer
class C extends AnyRef {
  def f: Tuple2[String, String] =
    Ar[String]("jan").->[String]("january")
}

// After erasure
class C extends Object {
  def f(): Tuple2 =
    ArrowAssoc.->$extension(Ar("jan"), "january")
}
```

No boxing!

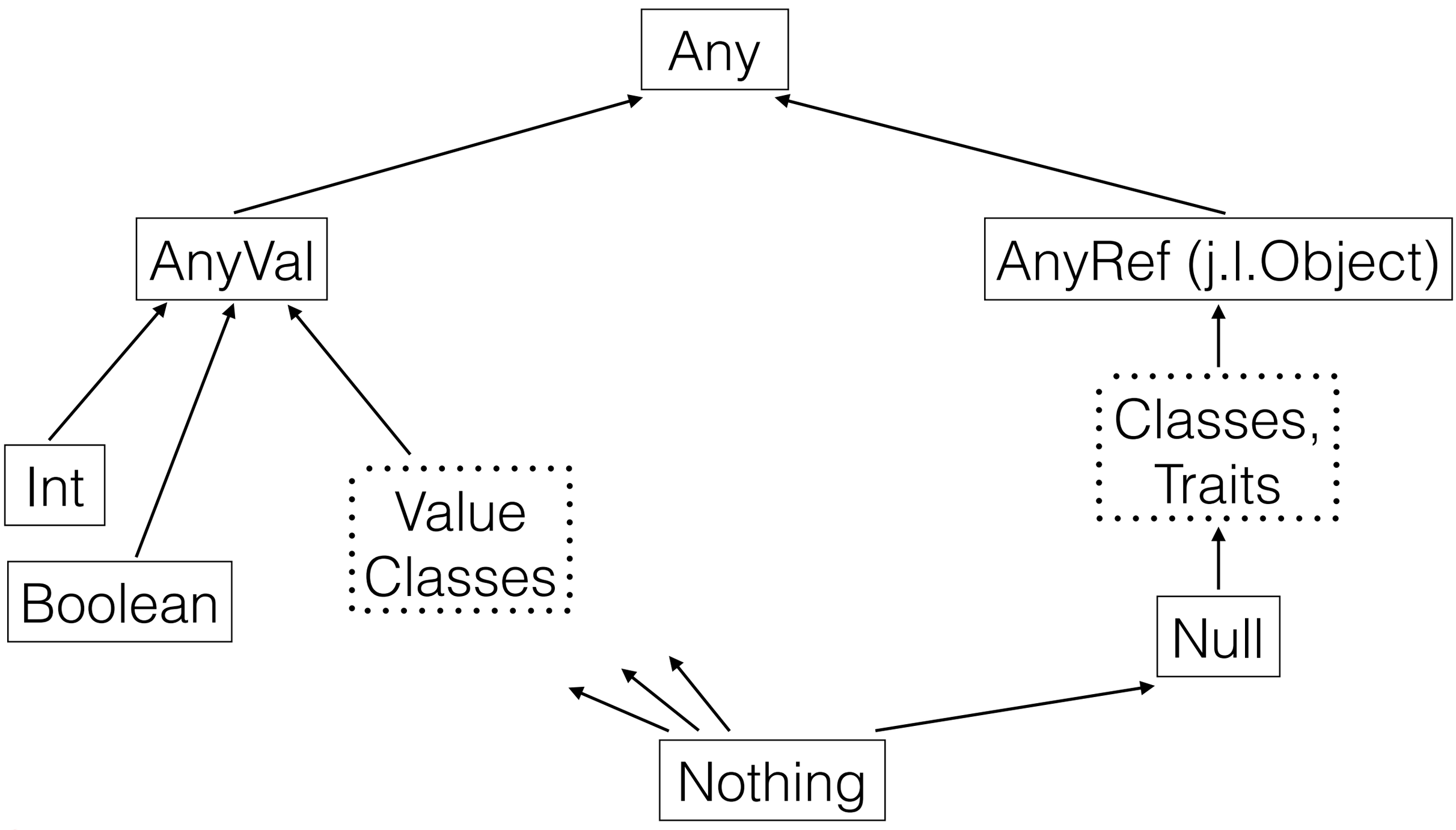
Agenda

- Introduction to value classes
- Compiler transformation: an example
- Value classes in Scala's type hierarchy
- Limitations and feature interactions

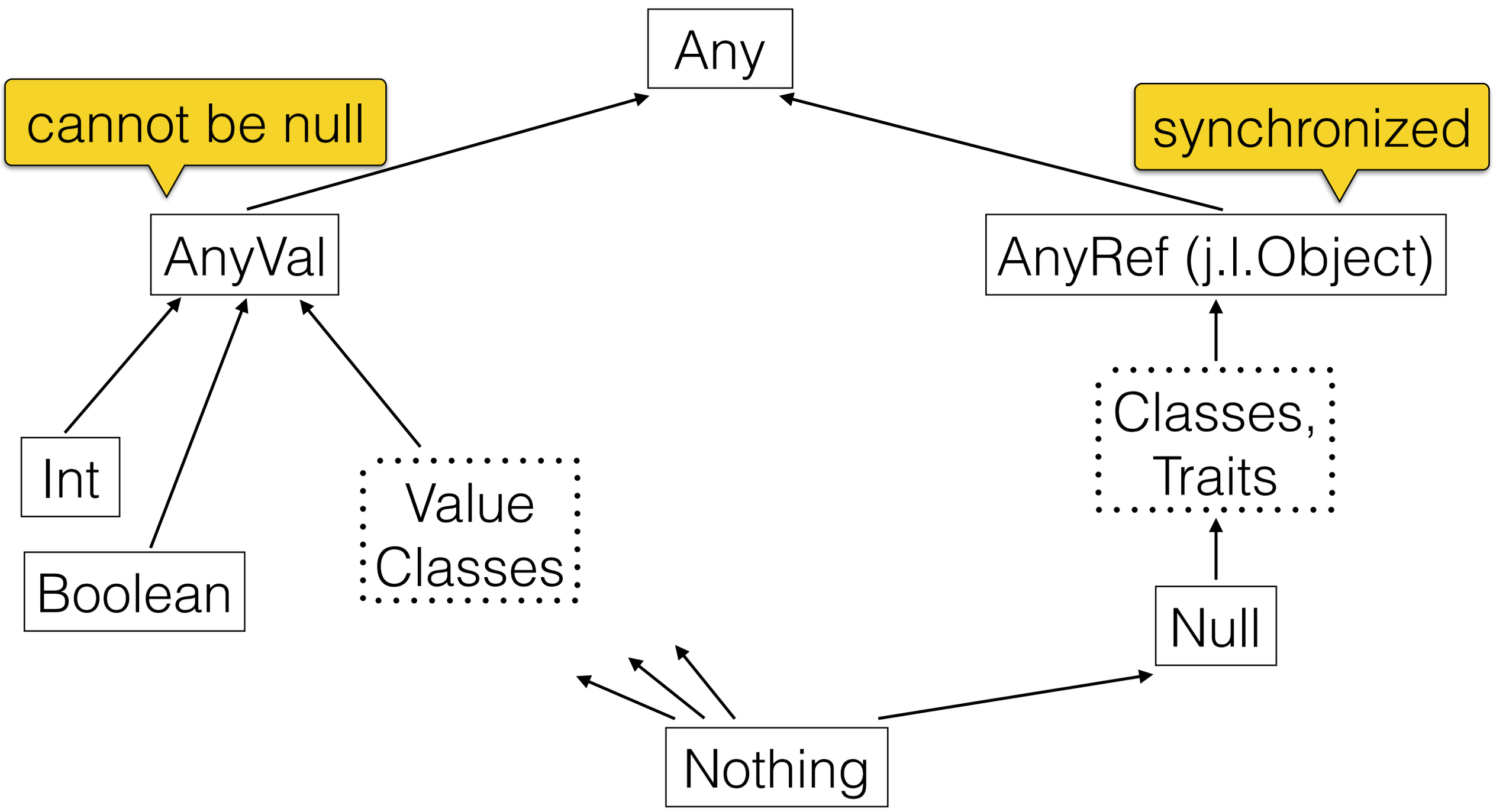
Defining Value Classes

- No fields, only (one single) class parameter
- No initializer statements
- The wrapped value may not be a value class
  - ↛ Not yet implemented
- `hashCode` and `equals` cannot be user-defined
- Value classes extend `AnyVal`

Type Hierarchy



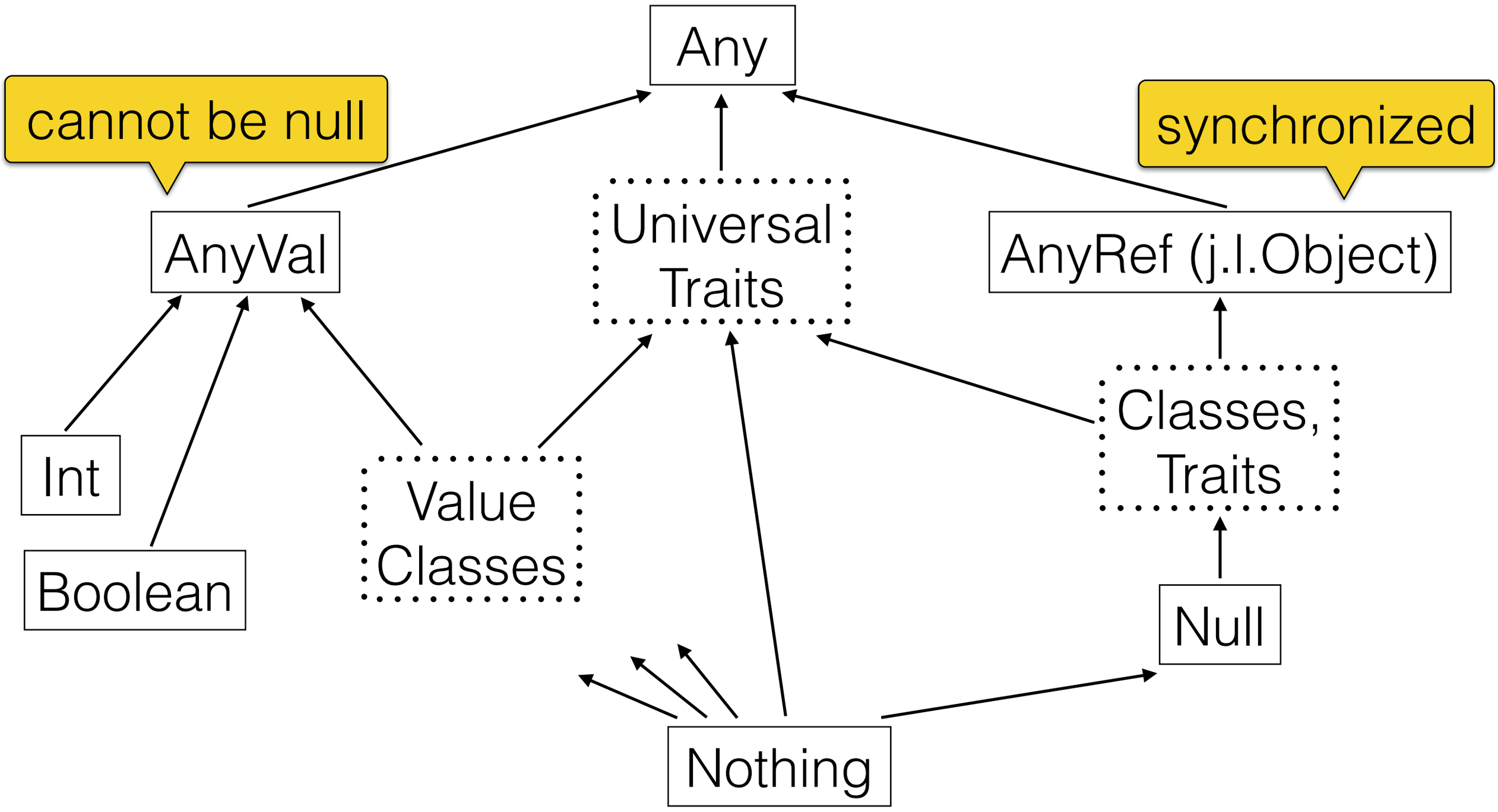
Type Hierarchy



Extending Traits

- Traits may have fields, initializers
  - ↠ Not suitable as parents for value classes
- Value classes can extend *universal* traits
  - ↠ Universal traits extend **Any**
  - ↠ No fields, no synchronization

Type Hierarchy



Agenda

- Introduction to value classes
- Compiler transformation: an example
- Value classes in Scala's type hierarchy
- Limitations and feature interactions

Boxing

- Assigning a value to a supertype (parent trait, `AnyVal`, `Any`)

- Generics: no specialization for value classes (yet)
  - ↣ For data structures *and* methods

- Values stored in arrays ☹
  - ↣ Need `meters.isInstanceOf[Array[Meter]]`

Trait Method Calls

```
trait Super extends Any {
  def x: Int
  def f = this.x
}
def t1(s: Super) = s.f
```

Needs to be a virtual call.
Requires an object.

```
class Value(val x: Int) extends AnyVal with Super
def t2(v: Value) = v.f
```

Code of `f` (compiled separately)
assumes existence of an instance

Performance Model

- Boxing is implicit: the code does not reveal the representation
- Difficult for users: requires knowledge about the limitations
- Scala favours uniformity in other places, e.g.,
  - while / for loops
  - Collections of primitives

Overloading Restrictions

```
class Meter(val x: Double) extends AnyVal
class Mile(val x: Double) extends AnyVal

trait Distance {
  def add(m: Meter): Distance
  def add(m: Mile): Distance
}

error: double definition:
  def add(m: Meter): Distance
  def add(m: Mile): Distance
have same type after erasure: (m: Double)Distance
```

Manifestation at Bridges

```
class C[T](val x: T) extends AnyVal

trait T[A] { def f: A }

class K extends T[C[Object]] {
  def f: C[Object] = ..
}
```

```
error: bridge generated for member
                  method f: ()C[Object] in class K
which overrides method f: ()A in trait T
clashes with definition of the member itself;
both have erased type ()Object
```

Extending Java Interfaces

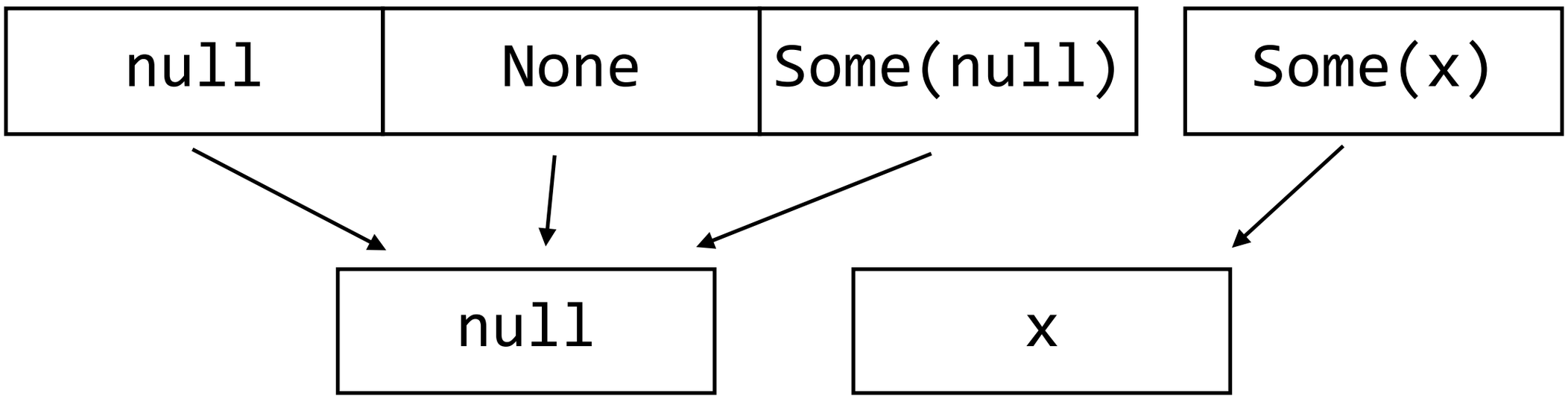
- Java interfaces in Scala are subtypes of `AnyRef`
- Making them *universal* (extend `Any`) is source-incompatible

```
def f(i: JavaIface) = i.synchronized { .. }
```

- Hack for `Serializable`, `Comparable`
  - ↣ Allow case classes to be value classes
  - ↣ Scala library has value classes extending `Comparable`

https://groups.google.com/forum/#!topic/scala-internals/12h2TgDFnDM
https://groups.google.com/forum/#!topic/scala-internals/jsVlJI4H5OQ

Option[T] value class?




```
scala> Set[String](null, "hi") find (_ == null)
res: Option[String] = Some(null)
```

https://groups.google.com/forum/#!topic/scala-language/Mz_VoJdJf1w

Feature Interactions in the Compiler

- Implementation in the compiler is difficult at the edges
- Examples from the issue tracker
  - ↠ VCs + type bounds: compiler crash (SI-6304)
  - ↠ VC method with lazy val: compiler crash (SI-6358)
  - ↠ VC with private[this] method: compiler crash (SI-7019)

Future: Multiple Fields

- Doable on today's VM
- Investment vs. return, given the limitations
  - Returns require boxing (or other tricks)
  - Boxing in arrays
  - Writes to fields are not atomic

Future

- Integrate with specialization
- Re-implement on top of the VM support
  - ↠ Profit: arrays of values, return multiple values
  - ↠ Compatibility with the rest of the ecosystem (e.g., reflection)