Example 1

Hello world (using scalac)

Hello world (using scalac)

example_1.scala

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

Hello world (using scalac)

example_1.scala

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scalac example_1.scala
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala Example1
Helloo world!
```

Invoke the scala compiler

Hello world (using scalac)

example_1.scala

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scalac example_1.scala
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala Example1
Helloo world!
```

Invoke the scala compiler to compile to bytecode

Hello world (using scalac)

example_1.scala

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scalac example_1.scala
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala Example1
Helloo world!
```

A class file will be created, execute it using the scala interpreter

Hello world (using scalac)

example_1.scala

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scalac example_1.scala
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala Example1
Helloo world!
```

Hello world (using scalac)

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

Here, `object` is a singleton object (not a class - more later!)

Hello world (using scalac)

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

main is the usual entry point into the code, as in java

Hello world (using scalac)

`main` is the usual entry point into the code, as in java

`App` is a trait that can be extended to achieve the same effect

Hello world (using scalac)

example_1a.scala

```
object Example1A extends App {
    println("Helloo world!");
}
```

App is a trait that can be extended to achieve the same effect

Hello world (using scalac)

example_1a.scala

```
object Example1A extends App {
    println("Helloo world!");
}
```

App is a trait that can be extended to achieve the same effect

Hello world (using scalac)

example_1a.scala

```
object Example1A extends App {
    println("Helloo world!");
}
```

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scalac example_1a.scala
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala Example1A
Helloo world!
```

Invoke the scala compiler as usual

Hello world (using scalac)

```
object Example1 {
  def main(args: Array[String]) {
    println("Helloo world!");
  }
}
```

main is the usual entry point into the code, as in java

Example 2

Hello world (in the REPL)

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

The REPL ("Read-Evaluate-Print-Loop") shell is the interactive mode for running Scala

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

Invoke scala at the REPL command line

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

Invoke scala at the REPL command line

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

Type out the println command..

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

instant gratification follows..

Hello world (in the REPL)

```
Vitthals-MacBook-Pro:scala_code_examples vitthalsrinivasan$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala> println("Hello world!")
Hello world!

scala> :quit
```

..and quit when done!

Example 3

Mutable and Immutable 'variables'

Mutable and Immutable

'variables'

'storage unit' is actually the term we ought to use..

An immutable variable is not really a variable..

but 'variable' is easier to understand!

Variables are defined using

val or var

val is similar to the final keyword in Java

val	Designates the variable as immutable
var	Designates the variable as mutable

val	immutable	The variable cannot be reassigned to a new value
var	mutable	The variable can be reassigned

Every 'variable' (storage unit) must be declared using one of these keywords

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

A Mutable variable

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

An Immutable variable

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

Specifying the type is optional

Scala is statically typed, but has powerful type inference

Scala is statically typed, but has powerful type inference

Specifying the type is optional

Scala is statically typed, but has powerful type inference

Specifying the type is optional

Scala is statically typed, but has powerful **type inference**

Specifying the **type** is optional

Scala is statically typed, but has powerful type inference

Specifying the type is optional

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

Specifying the type is optional

Scala is statically typed, but has powerful type inference

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> val PI : Double = 3.14
PI: Double = 3.14
```

var or val

```
scala> val PI : Double = 3.14
PI: Double = 3.14


scala> PI = 22/7
<console>:12: error: reassignment to val
       PI = 22/7
          ^
```

An Immutable variable

cannot be reassigned
a new value

var or val

```
scala> var radius : Double = 10
radius: Double = 10.0
```

```
scala> radius = 12
radius: Double = 12.0
```

A mutable variable

can be reassigned a new value

but not a new type!

var or val

Why does this distinction matter?

Immutability is an important concept in Scala

Scala encourages the use of immutable variables to drive side-effect free code

var or val

Why does this distinction matter?

immutable storage units also help keep code safe for concurrent/ distributed applications

var or val

Why does this distinction matter?

If you know that a variable's state should not change once it is assigned

declare it immutable

var or val

Why does this distinction matter?

This makes sure that there are no unintentional state changes to the variable

It makes maintaining code far easier

Example 4

Type Inference

Scala Type Inference

We will see the full power - and complexity - of Type Inference later..

..but for now here are some simple examples of how it works

Scala Type Inference

here are some simple examples of how it works

Variable types can often be omitted

Function return types can often be omitted

Polymorphic method calls and generic class instantiations can often be omitted

Variable types can often be omitted

```
scala> var radius = 10.0
radius: Double = 10.0
```

No type is specified for the variable `radius`

But Scala type inference sees the value of `10.0`

And infers the type is `Double`

Variable types can often be omitted

```
scala> var radius2 = 10
radius2: Int = 10
```

No type is specified for the variable `radius`

But Scala type inference sees the value of `10`

And infers the type is `Int`

Variable types can often be omitted

```
scala> var radius:Double = 10
radius: Double = 10.0
```

The assigned value is `10`

Scala type inference would infer this is an `Int`

to get around this, explicitly specify type `Double`

Variable types can often be omitted

```
scala> var radius:Int = 10.0
<console>:11: error: type mismatch;
 found   : Double(10.0)
 required: Int
       var radius:Int = 10.0
```

The assigned value is `10.0`

Scala type inference would infer this is an `Double`

But, the explicitly specified type is `Int`

Higher rank value with lower rank type? Result

`error: type mismatch`

Variable types can often be omitted

```
scala> var radius:Double = "10.0"
<console>:11: error: type mismatch;
 found   : String("10.0")
 required: Double
       var radius:Double = "10.0"
                           ^
```

Type is explicitly specified as **Double**

But the assigned value is a string **“10.0”**

The result is an **error: type mismatch**

Scala Type Inference

here are some simple examples of how it works

Variable types can often be omitted

Function return types can often be omitted

Polymorphic method calls and generic class instantiations can often be omitted

Scala Type Inference

here are some simple examples of how it works

Variable types can often be omitted

Why does type inference matter?

Function return types can often be omitted

Polymorphic method calls and generic class instantiations can often be omitted

Why does type inference matter?

Backstory:

Statically Typed Languages

Java, C, C++

Dynamically Typed Languages

Python, Javascript

Statically Typed Languages

Java, C, C++

The type of every variable is known at compile time

Bugs get caught quickly and easily

code is more verbose

Dynamically Typed Languages

Python, Javascript

The type of every variable is known only at run time

Lots of nasty bugs creep in

code is quick & dirty to write

Why does type inference matter?

Statically Typed Languages

Dynamically Typed Languages

Java, C, C++

Python, Javascript

The type of every variable is known at compile time

The type of every variable is known only at run time

Bugs get caught quickly and easily

Bugs can sneak deep in

code is more verbose

code is quick & dirty to write

Scala seeks to get the best of both worlds via Type Inference

Why does type inference matter?

Scala Type Inference

Scala is statically typed

but it has an elaborate type inference system to guess types

so Scala code often looks more like Python code rather than like Java code

Why does type inference matter?

Why does type inference matter?

Scala is statically typed

but it has an elaborate type inference system to guess types

so Scala code often looks more like Python code rather than like Java code

Example 5

String Operations

String Operations

Usual stuff

String interpolation using `s""`

printf notation using `f""`

String Operations

Usual stuff

String interpolation using `s""`

printf notation using `f""`

Usual String Operations

```
scala> val name = "Vitthal"
name: String = Vitthal

scala> val greeting = "Hello"
greeting: String = Hello

scala> greeting + name
res2: String = HelloVitthal

scala> greeting + "\n" + name
res3: String =
Hello
Vitthal
```

Usual String Operations

```
scala> val complicatedGreeting = """You are amazing,
     | incredible,
     | YUGE
     | and ever so gracious
     | """
complicatedGreeting: String =
"You are amazing,
incredible,
YUGE
and ever so gracious
"
```

You can create multi-line strings using triple quotes """

Usual String Operations

```
scala> val PI = "Pi"
PI: String = Pi

scala> val PI2 = "Pi"
PI2: String = Pi

scala> PI == PI2
res10: Boolean = true
```

Unlike in Java, it is safe to compare strings using ==

String Operations

Usual stuff

String interpolation using `s""`

printf notation using `f""`

String Operations

Usual stuff

String interpolation using s""

printf notation using f""

String interpolation using s""

```
scala> s"$greeting, $name, How are you today?"
res5: String = Hello, Vitthal, How are you today?
```

Preface the string with s

String interpolation using s""

```
scala> s"$greeting, $name, How are you today?"
res5: String = Hello, Vitthal, How are you today?
```

Preface the string with s

Then, the string can contain variables, delimited by $

String interpolation using s""

```
scala> s"$greeting, $name, How are you today?"
res5: String = Hello, Vitthal, How are you today?
```

Preface the string with **s**

Then, the string can contain variables, delimited by **$**

Scala will resolve those variable names in the output

String interpolation using s""

```
scala> s"${greeting*2}, $name, How are you today?"
res7: String = HelloHello, Vitthal, How are you today?
```

You can create formulae using {}

Here, for instance, the string repeats,
due to the *2

String Operations

Usual stuff

String interpolation using s""

printf notation using f""

String Operations

Usual stuff

String interpolation using s""

printf notation using f""

printf notation using f""

```
scala> val PI = 3.14159
PI: Double = 3.14159
```

```
scala> f"PI evaluates to $PI%.2f"
res9: String = PI evaluates to 3.14
```

Formatting strings is easy - preface the string with f

printf notation using f""

```
scala> val PI = 3.14159
PI: Double = 3.14159
```

```
scala> f"PI evaluates to $PI%.2f"
res9: String = PI evaluates to 3.14
```

Formatting strings is easy - preface the string with `f`

Variables can be interpolated using `$`

printf notation using f""

```
scala> val PI = 3.14159
PI: Double = 3.14159
```

```
scala> f"PI evaluates to $PI%.2f"
res9: String = PI evaluates to 3.14
```

Formatting strings is easy - preface the string with `f`

Variables can be interpolated using `$`

And the formatting can be specified using %

printf notation using f""

```
scala> val PI = 3.14159
PI: Double = 3.14159
```

```
scala> f"PI evaluates to $PI%.2f"
res9: String = PI evaluates to 3.14
```

And the formatting can be specified using %

String Operations

Usual stuff

String interpolation using s""

printf notation using f""

String Operations

Usual stuff

String interpolation using `s""`

printf notation using `f""`

Example 6

A Unified Type System

A Unified Type System

In Scala, all values are instances of a class (no exceptions)

But, since Scala runs on the JVM, the distinction between value and reference types must still exist somehow

But, since **Scala runs on the JVM**, the distinction between value and reference types must still exist somehow

But, since Scala runs on the JVM, the distinction between **value and reference types** must still exist somehow

But, since Scala runs on the JVM, the distinction between value and reference types must still exist somehow

But, since Scala runs on the JVM, the distinction between value and reference types must still exist somehow

Scala squares this circle by adding a new superclass, as well as superclasses for all value and reference types

A Unified Type System

In Scala, all values are instances of a class (no exceptions)

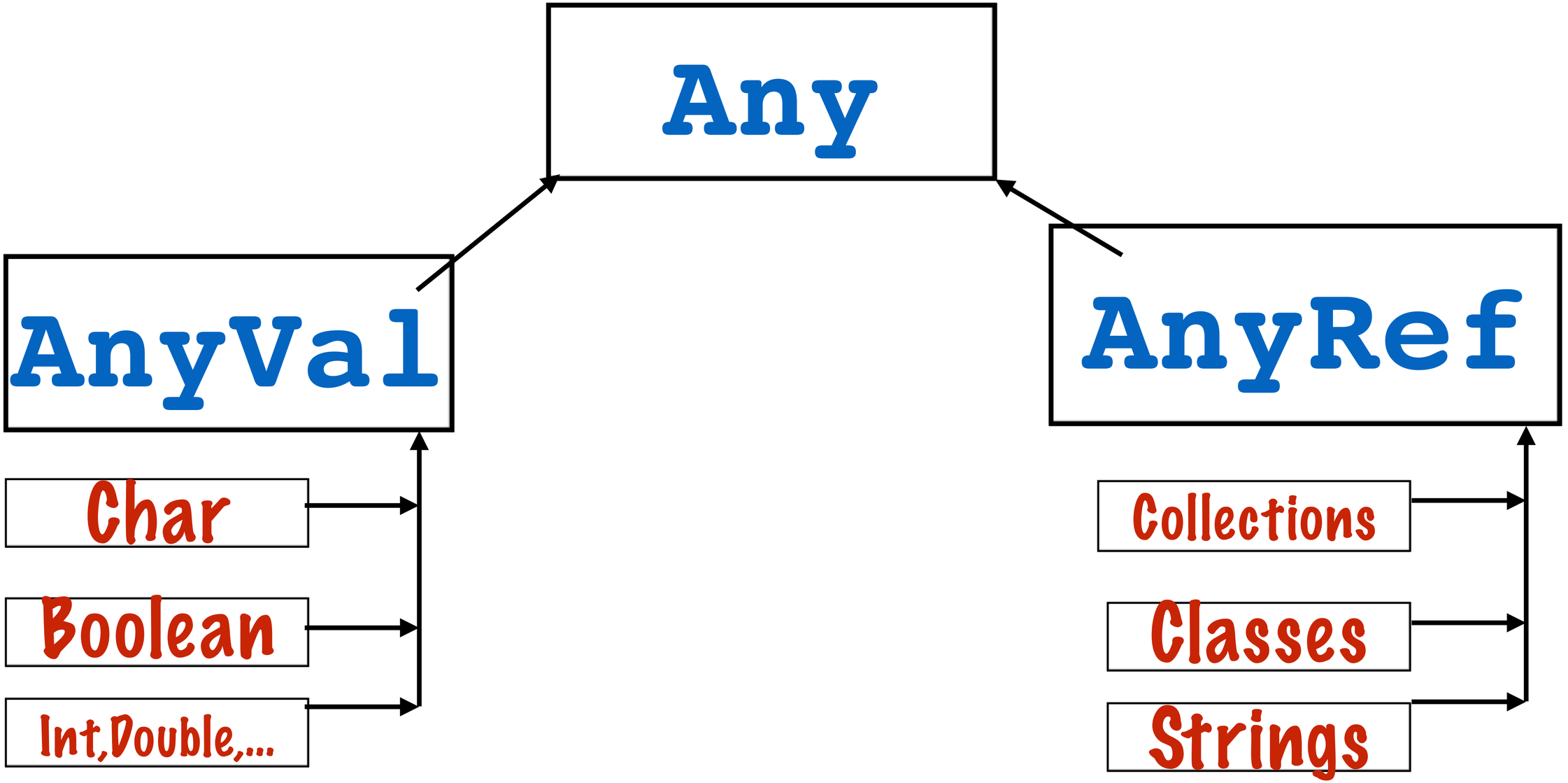
But, since Scala runs on the JVM, the distinction between value and reference types must still exist somehow

Scala squares this circle by adding a new superclass, as well as superclasses for all value and reference types

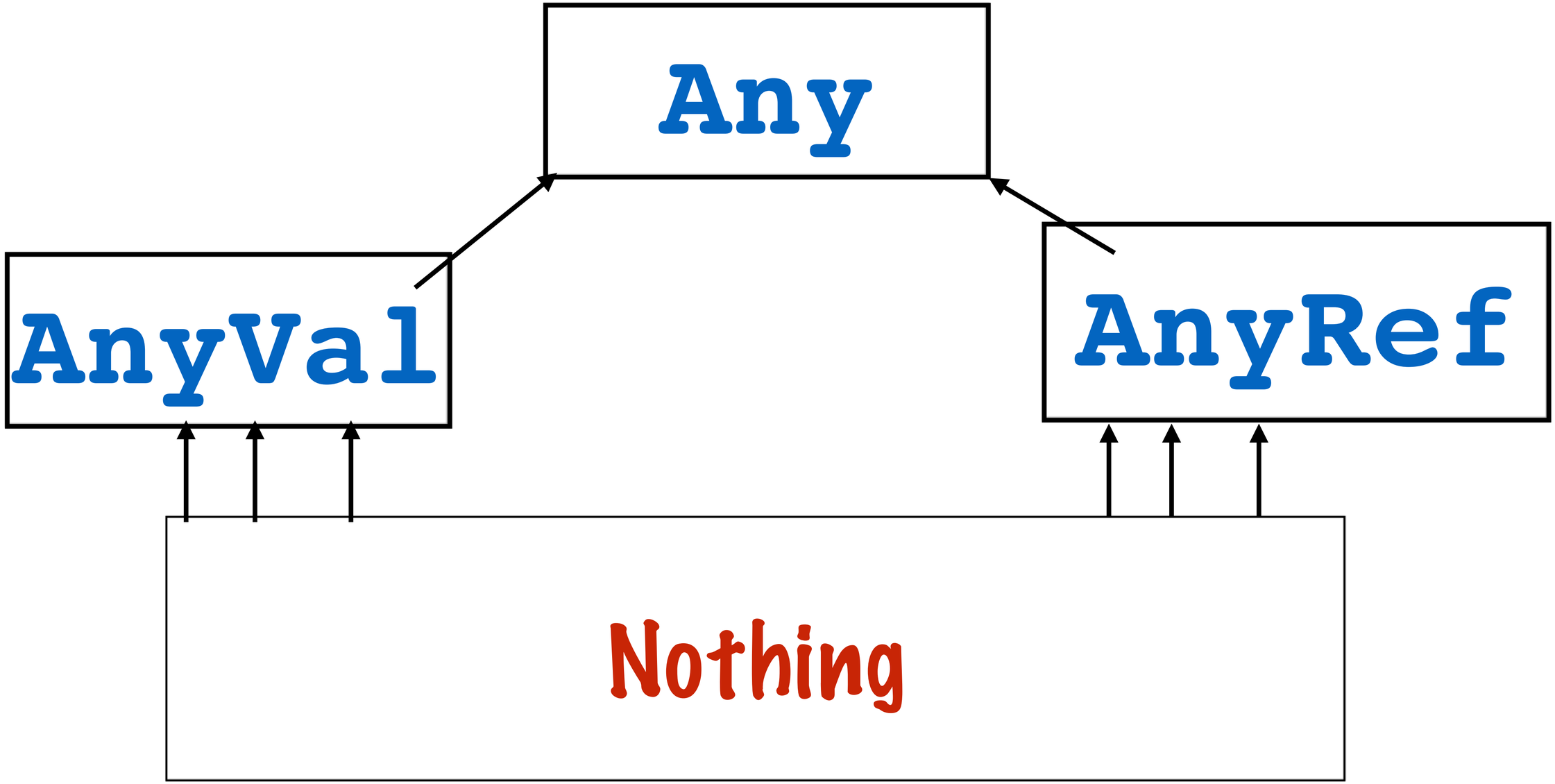
A Unified Type System

Scala squares this circle by adding a new superclass, as well as superclasses for all value and reference types

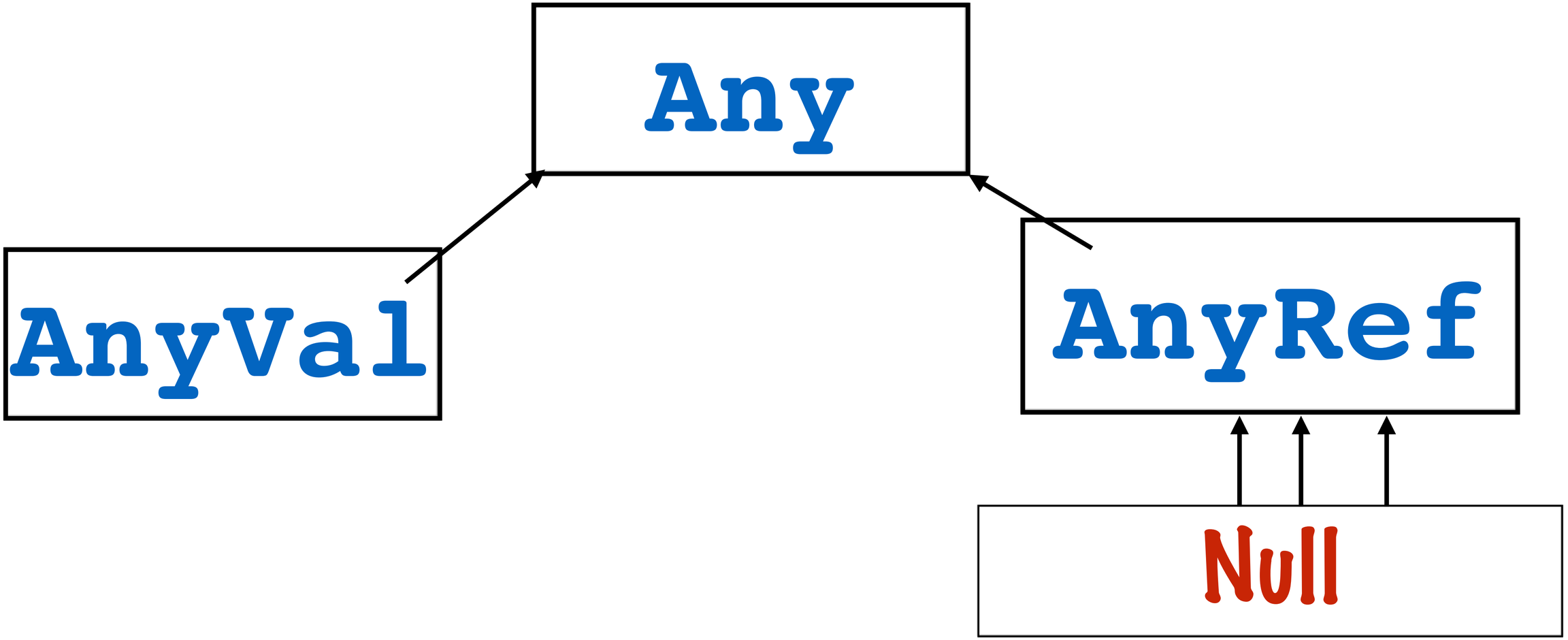
A Unified Type System



A Unified Type System



A Unified Type System



Consider 3 functions

```
scala> def printAny(x: Any) = println(x)
printAny: (x: Any)Unit

scala> def printAnyVal(y: AnyVal) = println(y)
printAnyVal: (y: AnyVal)Unit

scala> def printAnyRef(z: AnyRef) = println(z)
printAnyRef: (z: AnyRef)Unit
```

Consider 3 functions

```
scala> def printAny(x: Any) = println(x)
printAny: (x: Any)Unit
```

```
scala> def printAnyVal(y: AnyVal) = println(y)
printAnyVal: (y: AnyVal)Unit
```

```
scala> def printAnyRef(z: AnyRef) = println(z)
printAnyRef: (z: AnyRef)Unit
```

Take **Any** type, and print it

Consider 3 functions

Take `AnyVal` type, and print it

```
scala> def printAnyVal(y: AnyVal) = println(y)
printAnyVal: (y: AnyVal)Unit
```

```
scala> def printAnyRef(z: AnyRef) = println(z)
printAnyRef: (z: AnyRef)Unit
```

Consider 3 functions

Take `AnyRef` type, and print it

```
scala> def printAnyRef(z: AnyRef) = println(z)
printAnyRef: (z: AnyRef)Unit
```

Instantiate 2 values

```
scala> val someVal = 5
someVal: Int = 5

scala> val someRef = new Object
someRef: Object = java.lang.Object@1de0a46c
```

Subtype of `AnyVal`

Instantiate 2 values

```
scala> val someVal = 5
someVal: Int = 5

scala> val someRef = new Object
someRef: Object = java.lang.Object@1de0a46c
```

Subtype of `AnyRef`

Instantiate 2 values

```
scala> val someVal = 5
someVal: Int = 5

scala> val someRef = new Object
someRef: Object = java.lang.Object@1de0a46c
```

Java inter-op in action

```
scala> def printAny(x: Any) = println(x)
printAny: (x: Any)Unit
```

printAny will work with either of these values

```
scala> printAny(someVal)
5

scala> printAny(someRef)
java.lang.Object@1de0a46c
```

```
scala> def printAnyVal(y: AnyVal) = println(y)
printAnyVal: (y: AnyVal)Unit
```

printAnyVal will work only with the value type...

```
scala> printAnyVal(someVal)
5

scala> printAnyVal(someRef)
<console>:14: error: type mismatch;
 found   : Object
 required: AnyVal
Note that implicit conversions are not applicable because they are ambiguous:
 both method ArrowAssoc in object Predef of type [A](self: A)ArrowAssoc[A]
 and method Ensuring in object Predef of type [A](self: A)Ensuring[A]
 are possible conversion functions from Object to AnyVal
       printAnyVal(someRef)
                   ^
```

```
scala> def printAnyRef(z: AnyRef) = println(z)
printAnyRef: (z: AnyRef)Unit
```

printAnyRef will work only with the ref type...

```
scala> printAnyRef(someRef)
java.lang.Object@1de0a46c

scala> printAnyRef(someVal)
<console>:14: error: the result type of an implicit conversion must be more spec
ific than AnyRef
       printAnyRef(someVal)
                   ^
```

Example 7

Emptiness in Scala

Emptiness in Scala

`null`	`Null`	`Nothing`
`Nil`	`None`	`Unit`

Emptiness in Scala

`null`

basically, the same as
null in Java

null

basically, the same as null in Java

```
scala> val x:String = null
x: String = null

scala> if (x == null) println("null") else println("not null")
null
```

reference types can be null,
but value types can not

null

basically, the same as null in Java

reference types can be null,
but value types can not

```
scala> val z:Int = null
<console>:11: error: an expression of type Null is ineligible for implicit conve
rsion
       val z:Int = null
                   ^
```

Emptiness in Scala

`null`

`Null`

`Nothing`

`Nil`

`None`

`Unit`

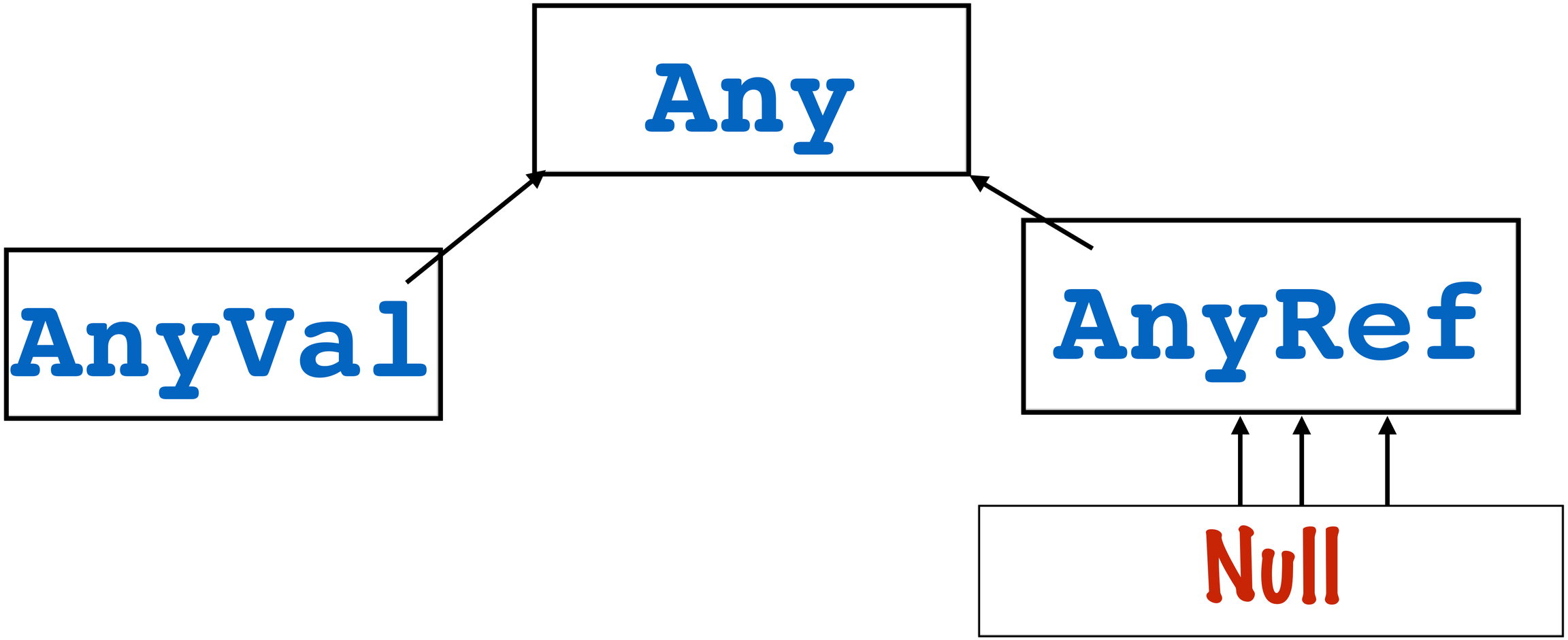
Emptiness in Scala

`Null`

`Null` is a trait (i.e. a type) not a value

`Null` is the type of `null`

`Null` is a trait (i.e. a type) not a value



Emptiness in Scala

`null`

`Null`

`Nothing`

`Nil`

`None`

`Unit`

Emptiness in Scala

Nothing

Nothing is a trait (i.e. a type) not a value

```
scala> val emptyList = List()
emptyList: List[Nothing] = List()
```

Nothing can never be instantiated..

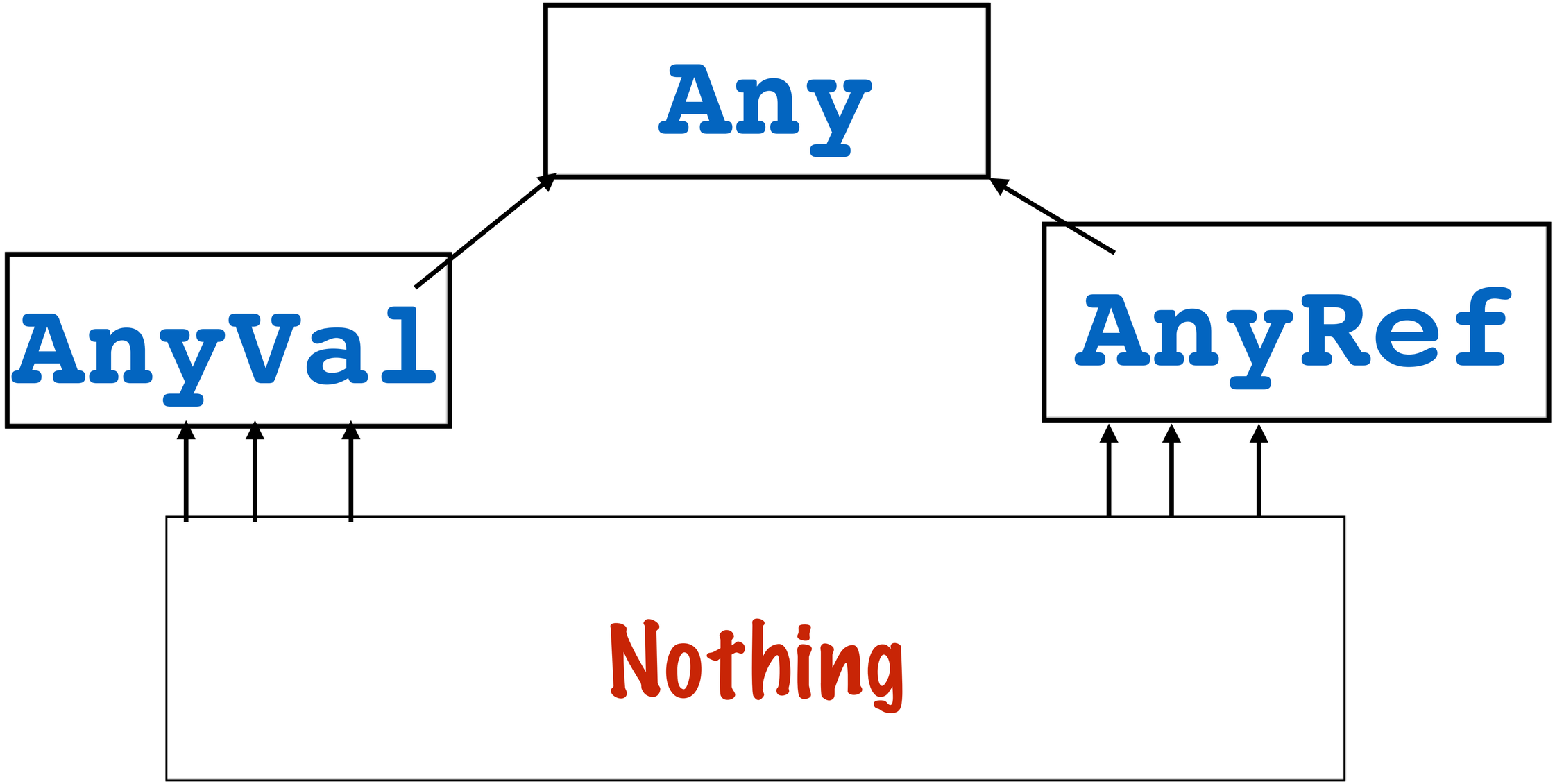
Emptiness in Scala

`Nothing`

`Nothing` can never be instantiated..

`Nothing` 'extends everything'

`Nothing` 'extends everything'



Emptiness in Scala


`null`

`Nil`


`Null`

`None`


`Nothing`

`Unit`

Emptiness in Scala

`Nil`

`Nil` is a special value associated with an empty `List`

`Nil` is a singleton instance of `List[Nothing]`

Emptiness in Scala

Nil

```
scala> val someList = List(1,2,3,4)
someList: List[Int] = List(1, 2, 3, 4)

scala> var listIter = someList
listIter: List[Int] = List(1, 2, 3, 4)
```

Lists are internally represented as linked lists, and use this special value to signify the end of the list

Emptiness in Scala

```
scala> val someList = List(1,2,3,4)
someList: List[Int] = List(1, 2, 3, 4)

scala> var listIter = someList
listIter: List[Int] = List(1, 2, 3, 4)
```

Nil

```
scala> while (listIter != Nil) {
     | println(listIter.head);
     | listIter = listIter.tail
     | }
1
2
3
4
```

Lists are internally represented as linked lists, and use this special value to signify the end of the list

Emptiness in Scala

- ✅ `null`
- ✅ `Null`
- ✅ `Nothing`
- ✅ `Nil`
- `None`
- `Unit`

Emptiness in Scala

`None`

`None` is a special value associated with an `Option`

`Option` is a (monadic) collection used to capture presence or absence of a value

None is a special value associated with an **Option**

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

Option is a (monadic) collection used to capture presence or absence of a value

`None` is a special value associated with an `Option`

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

`Option` is a (monadic) collection used to capture presence or absence of a value

`None` is a special value associated with an `Option`

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

`Option` is a (monadic) collection used to capture presence or absence of a value

`None` is a special value associated with an `Option`

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

```
scala> fraction(100,0)
res5: Option[Double] = None
```

`Option` is a (monadic) collection used to capture presence or absence of a value

`None` is a special value associated with an `Option`

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

`Option` is a (monadic) collection used to capture presence or absence of a value

`None` is a special value associated with an `Option`

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
     | = {
     | if (denom == 0) None
     | else Option(numer/denom)
     | }
fraction: (numer: Double, denom: Double)Option[Double]
```

```
scala> fraction(22,7)
res4: Option[Double] = Some(3.142857142857143)
```

`Option` is a (monadic) collection used to capture presence or absence of a value

Emptiness in Scala

- ✅ `null`
- ✅ `Null`
- ✅ `Nothing`
- ✅ `Nil`
- ✅ `None`
- `Unit`

Emptiness in Scala

`Unit`

`Unit` is basically like `void` in Java

`Unit` is the return type of a function that returns nothing, for instance

`Unit` is basically like `void` in Java

```
scala> def printAny(x:Any) {println(x)}
printAny: (x: Any)Unit
```

`Unit` is the return type of a function that returns nothing, for instance

Emptiness in Scala

- `null`
- `Null`
- `Nothing`
- `Nil`
- `None`
- `Unit`

Example 8

Type Operations

Type Operations

```
asInstanceOf

                isInstanceOf

  to<Type>

                getClass
```

Type Operations

```
asInstanceOf

                isInstanceOf

  to<Type>

                getClass
```

asInstanceOf

```
scala> 123.asInstanceOf[Long]
res16: Long = 123

scala> 123.24.asInstanceOf[Long]
res17: Long = 123
```

asInstanceOf

```
scala> "123.24".asInstanceOf[Long]
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Long
  at scala.runtime.BoxesRunTime.unboxToLong(BoxesRunTime.java:105)
  ... 32 elided
```

Type Operations

```
asInstanceOf

                isInstanceOf

  to<Type>

                getClass
```

to<Type>

```
scala> 123.toLong
res24: Long = 123

scala> 123.24.toLong
res25: Long = 123

scala> "123".toLong
res26: Long = 123
```

to<Type>

```
scala> "123".toLong
res49: Long = 123

scala> "abc".toLong
java.lang.NumberFormatException: For input string: "abc"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Long.parseLong(Long.java:589)
  at java.lang.Long.parseLong(Long.java:631)
  at scala.collection.immutable.StringLike$class.toLong(StringLike.scala:276)
  at scala.collection.immutable.StringOps.toLong(StringOps.scala:29)
  ... 32 elided
```

Type Operations

```
asInstanceOf

                isInstanceOf

  to<Type>

                getClass
```

isInstanceOf

```
scala> 123.isInstanceOf[Long]
res29: Boolean = false

scala> 123.toLong.isInstanceOf[Long]
res30: Boolean = true

scala> 123.isInstanceOf[Any]
res31: Boolean = true
```

isInstanceOf

```
scala> "123".isInstanceOf[Any]
res33: Boolean = true

scala> "123".isInstanceOf[AnyRef]
res34: Boolean = true
```

isInstanceOf

```
scala> null.isInstanceOf[Null]
<console>:12: error: type Null cannot be used in a type pattern or isInstanceOf
test
       null.isInstanceOf[Null]
                         ^


scala> 123.isInstanceOf[AnyVal]
<console>:12: error: type AnyVal cannot be used in a type pattern or isInstanceO
f test
       123.isInstanceOf[AnyVal]
                         ^
```

Type Operations

```
asInstanceOf

                isInstanceOf

  to<Type>

                getClass
```

getClass

```
scala> 123.getClass
res39: Class[Int] = int

scala> 123.toLong.getClass
res40: Class[Long] = long

scala> "123".getClass
res41: Class[_ <: String] = class java.lang.String

scala> List(12,3).getClass
res42: Class[_ <: List[Int]] = class scala.collection.immutable.$colon$colon
```