Introduction To Scala

CSC5003

Goals of This Lecture

- Discover the fundamentals of Scala. We will overview what is important for this course, you can practice more on your side.
- Understand how functional programming works in Scala.
- Get the basic functions for our future Big Data applications.

At the end of this lecture, you will know in Scala

- What the basic types are
- How to declare variables and what is the difference with a value
- How to declare a function and an anonymous function
- The basic data structures
- The control structures
- The most common high-order functions and how they replace usual control structures.

What Is Scala?

- Object-oriented + functional language
- Strongly typed
- Runs on the JVM (compatible with Java)
- Can be compiled to Javascript
- Very popular for Big Data applications

/!\ Scala 2 and Scala 3 have some differences in the syntax!

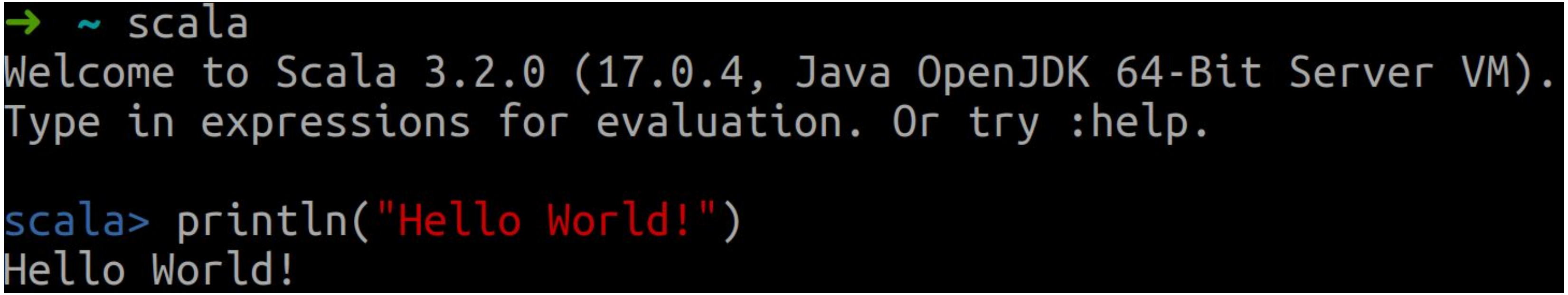
Scala 3 looks more like Python.

Scala 2 should still work in Scala 3.

https://scala-lang.org

Programming In Scala

- In your usual Java IDE + Scala plugin
- Using the Scala interpreter



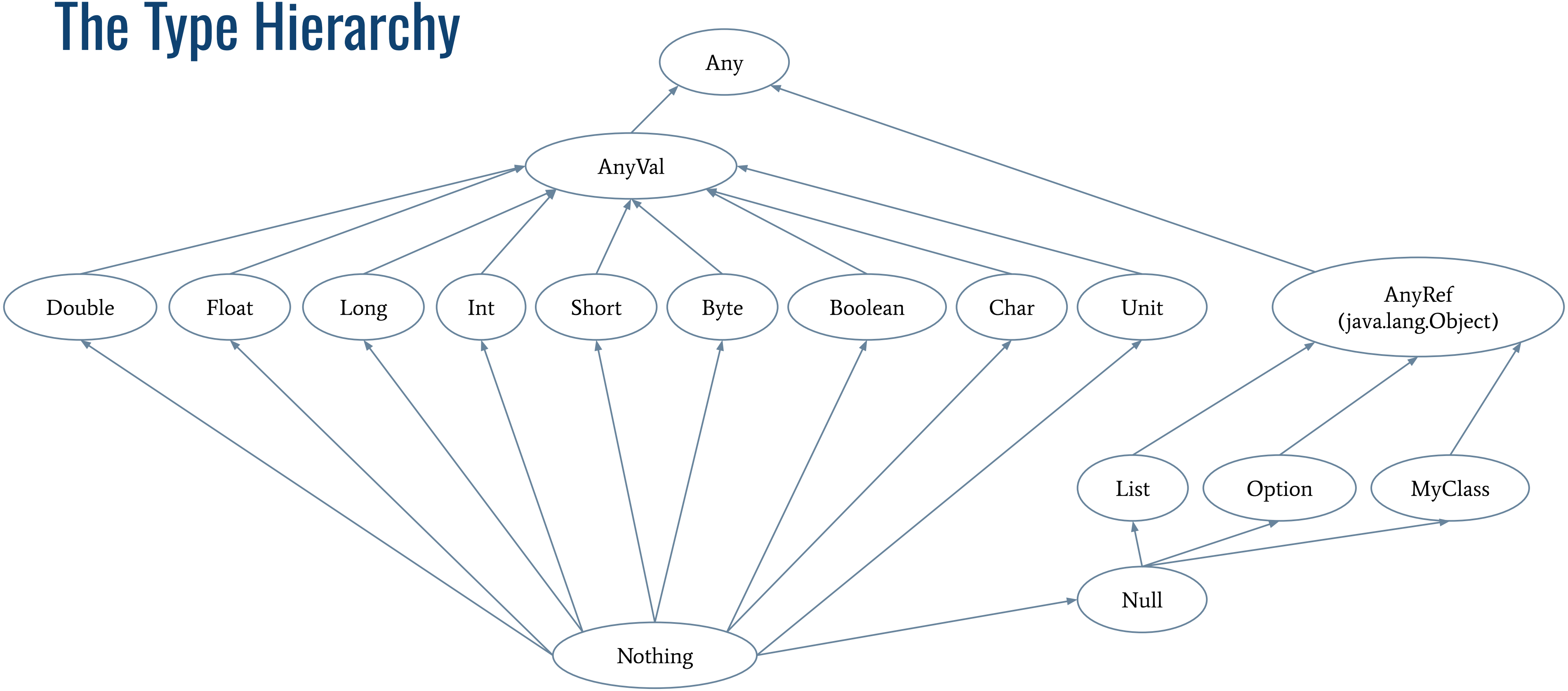
```
➜  ~ scala
Welcome to Scala 3.2.0 (17.0.4, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> println("Hello World!")
Hello World!
```

- SBT is the traditional build tool (similar to Maven and Gradle)
- More about this in the lab

Types and Variables

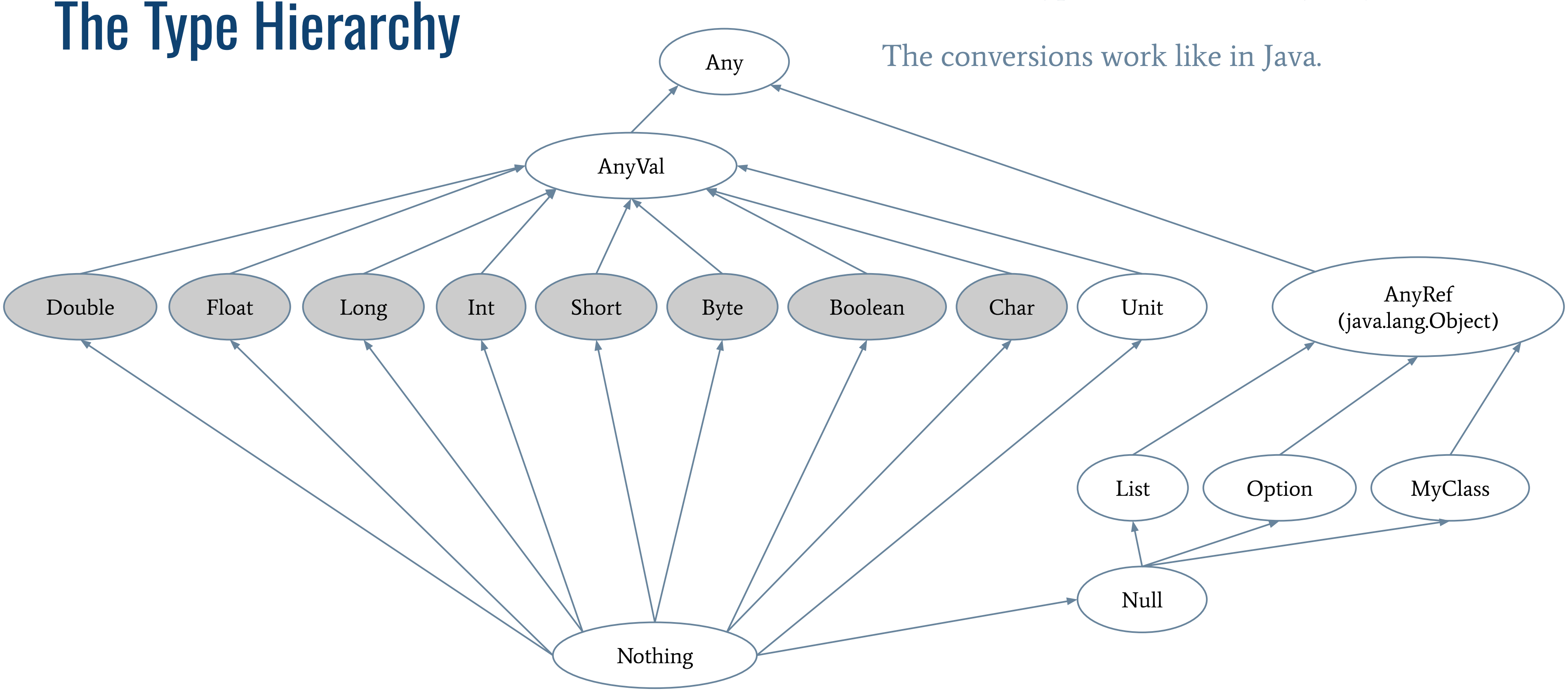
The Type Hierarchy



The Type Hierarchy

Traditional types in Java, but only Object Version!

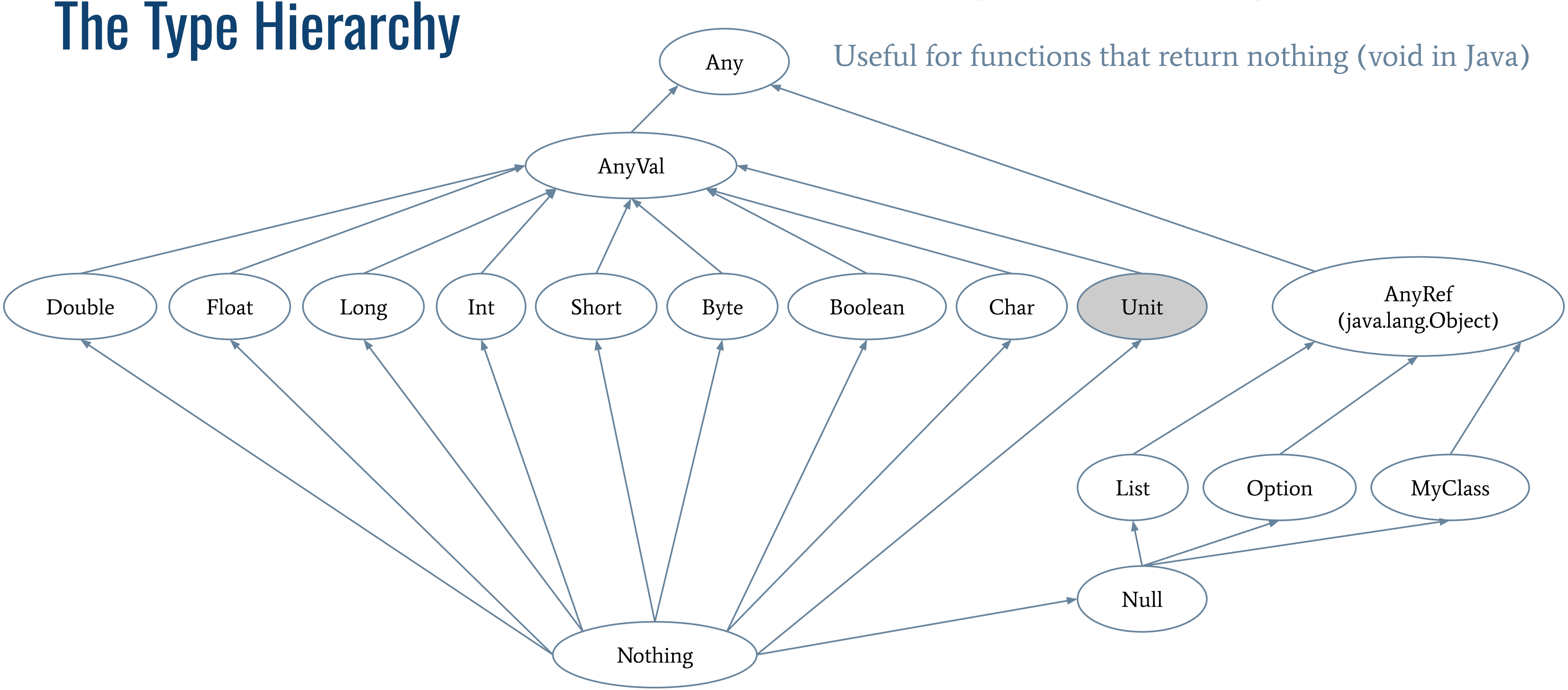
The conversions work like in Java.



The Type Hierarchy

Unit is a type that can have a single value: ()

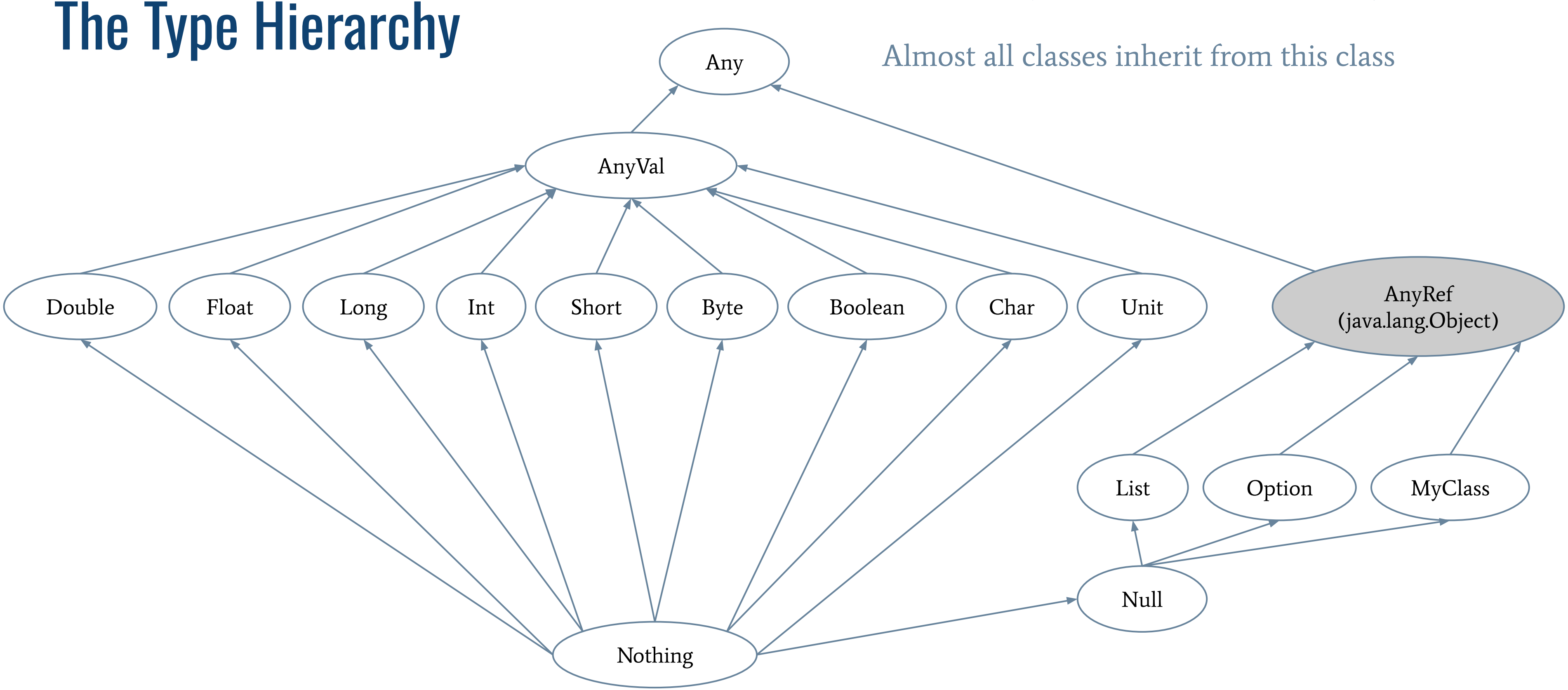
Useful for functions that return nothing (void in Java)



The Type Hierarchy

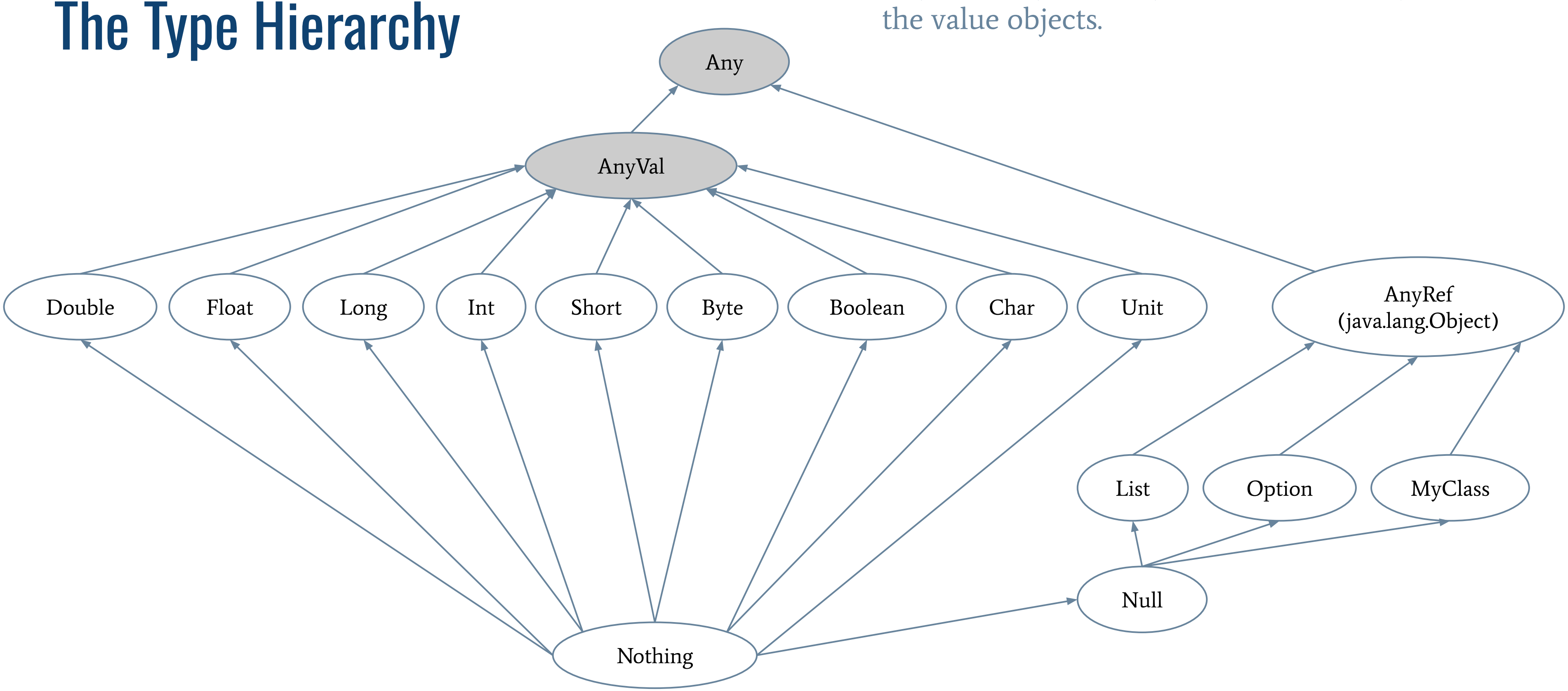
Our old Object in Java, now called AnyRef

Almost all classes inherit from this class



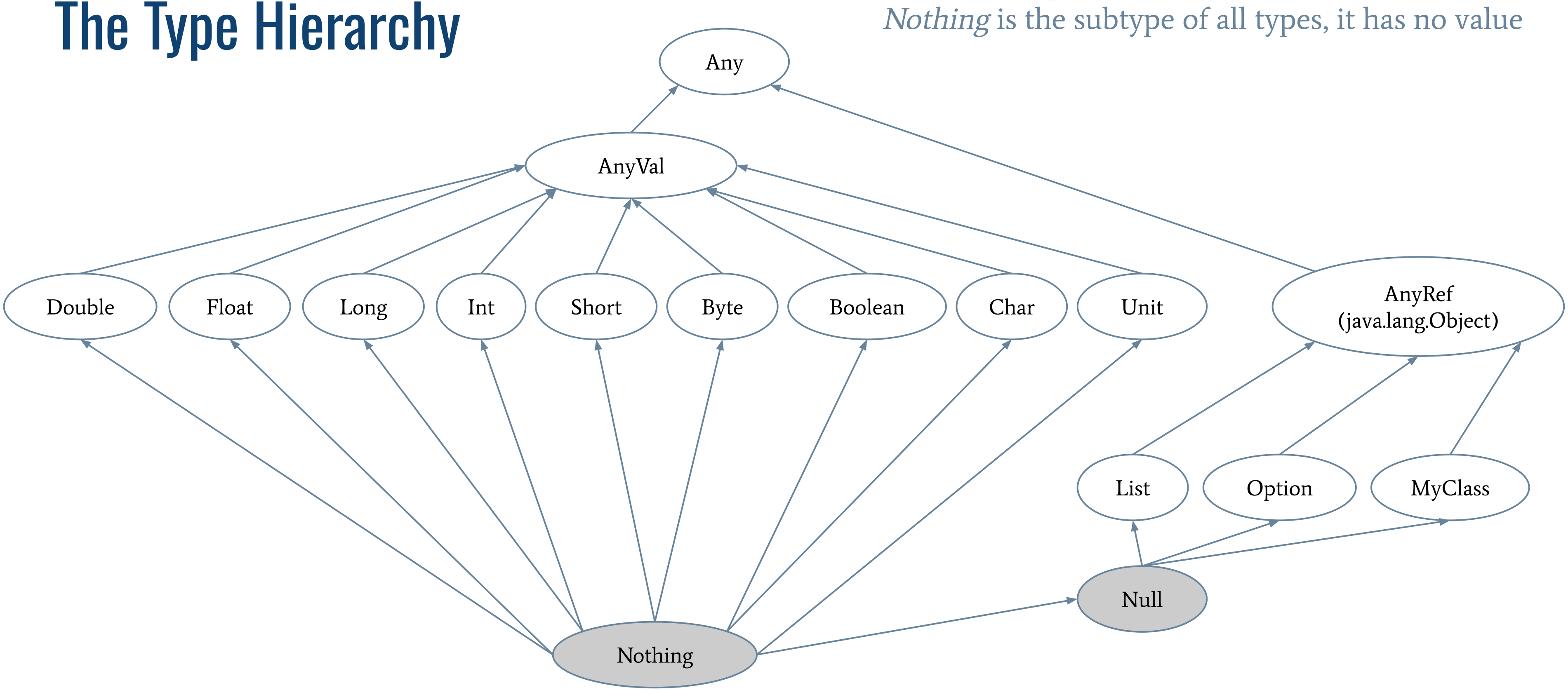
The Type Hierarchy

Any and *AnyVal* unify the traditional objects and the value objects.



The Type Hierarchy

Null is the type of the Java's *null* value
Nothing is the subtype of all types, it has no value



Values And Variables

- A Java-like variable can be declared with:
  - `var name: Type = value`
- A value is a variable that cannot change its content (final in Java)
  - `val name: Type = value`
  - Compatible with functional programming spirit where everything is immutable

Note: Semi-columns (;) are optional in Scala if you go to a new line.

Explicit VS Implicit Types

- Like in Java, we can explicitly specify the types of the variables
- However, when it is obvious, Scala can guess it.

```
val x = 3  // Here Scala guesses that x is an Int
> val x: Int = 3


val x: Long = 3 // We want to force Scala to consider x as a Long
> val x: Long = 3
```

In what follows, we might omit the type if Scala can infer it.

Functions

Functions

The declaration of a function works as follows:

```
def functionName(param1: Type1, param2: Type2, …): ReturnType = {
    // function content
}
```

Notes:

- The = is mandatory
- In Scala 3, the brackets ({}) are optional and can be, like in Python, replaced by indentations.

Functions - Examples

```
def mean(a: Int, b: Int): Double = {
    val c = a + b
    c / 2.0
}

mean(1, 2)
> 1.5

mean
> val res: (Int, Int) => Double = Lambda$1464/0x000000080114…
```

The **return statement is optional**
The last value is returned by default

Functions are objects like any others!
Note how the type is constructed

Functions - Examples

```
def sayHello(): Unit = {
    println("Hello")
}
```

Nothing to return here
println also return *Unit*

```
sayHello
> Hello
```

When there is no argument, we do not write the parenthesis

```
var sayHello2: () => Unit = sayHello

sayHello2
> Hello
```

Function can be manipulated like other objects!

Functions - Examples - Curryfication

```
def addAndMultiply(a: Int, b:Int)(multiplier: Float): Double = {
      (a + b) * multiplier
 }
```

A function can take several set of parameters!

```
addAndMultiply
> val res5: (Int, Int) => Float => Double = Lambda$1641/0x000…

addAndMultiply(1, 2)(3)
> 9

val time5 = addAndMultiply(2, 3)
> val time5: Float => Double = Lambda$1643/0x0000

time5(4)
> 20.0
```

We need to understand that *addAndMultiply* takes a pair of *Int* as input and outputs a function that transforms a *Float* into a *Double*

We can generate new functions from the original one

Functions - Examples

```
def apply(a: Int, b: Int, f: (Int, Int) => Any): Any = {
    f(a,b)
 }

apply(1, 2, mean)
> val res6: Any = 1.5
```

Here, we do not make assumption about the return type: it can be *anything*

With functional programming, it is common to pass other functions as argument

Anonymous Functions

- Functions are so common that we have fast ways to declare them:
  - `(param1: Type1, param2: Type2, …) => {content}: ReturnType`
- Examples:
  - `(x: Int) => {val y = 2; x + y}` `:Double` // Here, we force the return type to be a Double, not an Int
  - `(x: Int) => x + 1` // the brackets are often optional
- Similar to lambda functions in Python.
- **Placeholder syntax**: When the parameters are used only once, we can use placeholders. They allow us to not specify the parameters.
  - `f(_, _, _, …)` is equivalent to `(var1, var2, var3, …) => f(var1, var2, var3)`
  - e.g.: `_ + _` is equivalent to `(x, y) => x + y`
  - We can also specify the types if they cannot be inferred automatically: `(_ : Int) + (_: Int)`

Data Structures In Scala

Scala Data Structures

- You have a list of Scala data structures at https://www.scala-lang.org/api/3.2.0/
- There are two kinds of data structures:
  - ***Immutable***: Once initialized, they cannot be modified. In *scala.collection.immutable*.
  - ***Mutable***: Modifiable at will. In *scala.collection.mutable*.
- Some data structures are imported by default, but in their immutable form (`List`, `Set`, `Map`, ...)
- Otherwise, we import them with:
  - `import scala.collection.immutable.ListSet`

Note: The `new` keyword exists in Scala but is often omitted for the standard data structure. To learn more about this, see Scala's classes and companion objects.

Useful Data Structures - Tuples

```
// Initialization
val hostPort = ("localhost", 80)

//Access the components
hostPort._1
> localhost

hostPort._2
> 80
```

Useful Data Structures - Array

```
// An array initialized with null values
var arr: Array[String] = new Array[String](3)

// We can also initialize all the values
var arr2: Array[Int] = Array(1, 2, 3)

// /!\ Values are accessed using parenthesis, not square brackets []
arr(1) = "Hello"

// We can call functions on it, like in Java
arr.size
> 3
```

Useful Data Structures - Lists

```
val l0: List[Int] = List(1, 2, 3)  // Initialization

l0.head  // First element
> 1

l0.tail // All elements but the first one
> List(2, 3)

val l1 = 0 :: l0  // Append an element (immutable => creates a new list!)
> List(0, 1, 2, 3)

val l2 = l0 ::: l1  // Concatenate two lists (creates a new list!)
> List(1, 2, 3, 0, 1, 2, 3)
```

Useful Data Structures - Sets

```
// Initialization
val fruit = Set("apple", "orange", "peach", "banana")

// Check if contains an element
fruit.contains("apple")
> true

// Creates a new Set with an additional element
val newFruits = fruit + "pear"
> Set(peach, banana, orange, apple, pear)

// Creates a new Set and removes an element
val newerFruits = fruit - "apple"
> Set(orange, peach, banana)
```

Useful Data Structures - Map

```scala
// Initialization
val myMap: Map[String, Int] = Map("x" -> 1, "y" -> 2)

// Try to get the element associated to x, or return a default value
myMap.getOrElse("x", -1)
> 1

// Same with z
myMap.getOrElse("z", -1)
> -1

// Add an element to the Map. Remember, the structure is immutable!
myMap + ("z" -> 3)
> Map(x -> 1, y -> 2, z -> 3)
```

Useful Data Structures - Option

Option appears when a value could possibly not exists. In Java, we tend to return a null value. Although Scala also accepts the null value, it is more common to use Option. It prevents to have NullPointerException and forces the programmer to deal with it.

```
// When we access an element of a Map, we cannot be sure it
exists
myMap.get("x")
> val res2: Option[Int] = Some(1)

myMap.get("t")
> val res3: Option[Int] = None
```

Useful Data Structures - Option - Accessing Value

We could use `.get` or `.getOrElse` to access the content of an `Option`. However, Scala provides a better mechanism: **Pattern matching**.

```
myMap.get("t") match {
    case Some(v) => print(v)
    case None => print("Key not found")
 }
> Key not found
```

Pattern matching can also be used in other contexts (not seen in this course)

Useful Data Structures - Range

Represents all integers between two limits. Similar to Python.

```
// Get all integer between 1 and 10 (excluded)
Range(1, 10)
// or
1 until 10

// We can also have a step
Range(1, 10, 2)
//or
1 until 10 by 2
```

Control Structures

Conditions

- Similar to Java in Scala2 (in Scala3, similar to Python)

```
if (x < 0) {
    println("negative")
} else if (x > 0) {
    println("positive")
} else {
    println("zero")
}
```

- **Important**: The `if/else` structure always returns a value

```
val x = if (a < b) a else b
```

For Loops

A `for` loop always iterate through a collection.

```
// We use a Range for traditional loops
for (i <- 0 until 2) {
    println(i)
}
> 0
> 1
// We can also directly iterate over the elements of another collection
for (i <- fruit) {
    println(i)
}
> apple
> orange
> peach
> banana
```

For + If

We can combine for and if into a single expression, like in Python list comprehension.

```
for (i <- 0 until 4 if i%2 == 0){
    println(i)
}
> 0
> 2
```

For Expressions

Scala can use the `for` construction to create collections. A similar construction exists in Python.

```
for (f <- fruit if f.length > 4) yield f
> val res20: Set[String] = Set(apple, orange, peach, banana)
```

The return is inferred from the collection we are iterating from (either using the same type or an inherited type).

While Loop

Like in Java.

```
while (x >= 0) { x = f(x) }
```

We Try To Avoid These Structures

When we are writing a functional program, we try to avoid these structures.

- A loop can be replaced by **recursive functions** or **higher-order functions**

Why ?

- For are mainly used to manipulate mutable objects, often avoided in functional programming
- Focus on functions
- Improve reusability
- Declarative vs imperative = “what” vs “how”
  - E.g.: Get all even numbers between 0 and 10.
  - Declarative: I define what is an even number and gives this information to someone else.
  - Imperative: I have to do everything, i.e. read each number and check if it is even.

Common High-Order Functions In Scala

What Is a High-Order Function?

It is a function that takes one or several functions as argument, or a function that outputs a function.

They are opposed to *first-order functions*.

Example: The derivative function.

In Scala, some high-order functions are very common.

Filter Function

The `filter` function only takes elements in a collection that satisfy a given property.

```
// For List[A]
def filter(p: (A) => Boolean): List[A]

Range(1, 10).filter((x: Int) => x%2 == 0)
> val res13: IndexedSeq[Int] = Vector(2, 4, 6, 8)
```

Foreach Function

The `foreach` function applies a given function to all the elements in a collection.

```
// For List[A]
def foreach[U](f: (A) => U): Unit
```

```
Range(0, 3).foreach(println)
0
1
2
```

Map Function

The `map` function transforms elements of a given collection.

```
// For List[A]
def map[B](f: (A) => B): List[B]
```

```
Range(0, 3).map((x: Int) => x * 2)
> Vector(0, 2, 4)
```

Zip Function

The zip function stick together two collections into a new collection where the n^th^ item is the tuple of the old n^th^ items.

```
// For List[A]
def zip[B](that: IterableOnce[B]): List[(A, B)]

List("a", "b", "c").zip(Range(0, 3))
> List((a,0), (b,1), (c,2))
// or
List("a", "b", "c").zipWithIndex
```

Fold/FoldLeft/FoldRight Function

The `fold/foldLeft/foldRight` functions merge all the elements of the collection into a single value by “folding” the value using an accumulator.

Informal example: [1, 2, 3].fold(acc)(f) = f(f(f(acc, 1), 2), 3)

- foldLeft = folding from left to right
- foldRight = folding from right to left
- fold = folding with no presuposed order

```
def foldLeft[B](z: B)(op: (B, A) => B): B
def foldRight[B](z: B)(op: (A, B) => B): B

Range(0, 10).foldLeft(0)((x, y) => x+y)
> val res0: Int = 45
```

Reduce/ReduceLeft/ReduceRight Function

The `reduce/reduceLeft/reduceRight` functions work like the `fold` functions, except the initial value of the accumulator is computed from the two first elements.

```
def reduceLeft[B >: A](op: (B, A) => B): B
```

```
Range(0, 10).reduce((x, y) => x+y)
> 45
```

GroupBy

The function groupBy will group the elements of a collection using a key generated by a given function.

```
def groupBy[K](f: A => K): Map[K, C]

var l = List(1, 2, 3, 4, 5)

l.groupBy(_%2)

> HashMap(1 -> List(1, 3, 5), 0 -> List(2, 4))
```

Flatten Function

If we have an iterable collection containing iterable collections (a list of list for example), the `flatten` function merge all these iterables into a single one.

```
// For List[A]
def flatten[B](~~implicit toIterableOnce: (A) =>~~
~~IterableOnce[B]~~): List[B]

List(Range(0, 3), Range(3, 5)).flatten
> val res4: List[Int] = List(0, 1, 2, 3, 4)
```

What’s Next ?

- Scala is a complex language. We cannot see everything in this lecture, we will have to learn by yourself if you want to know more.
    - Object-oriented programming
    - Pattern matching
- Let’s go to the lab!