Day 1: Essentials

Today

Scala fundamentals

- absolute basics: values, expressions, type inference
- functions and recursion
- object-orientation: classes, objects, inheritance
- generics basics
- functional programming essentials
- collections overview

Intros & Admin

The Instructor

Daniel Ciocîrlan

Scala & Spark Engineer & Instructor

35000+ engineers

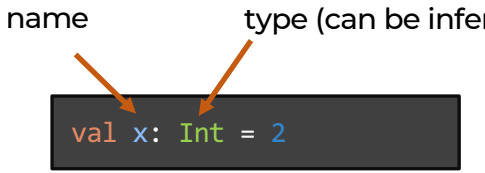
Specialities: Scala, Akka, Spark et. al.

rockthejvm.com

daniel@rockthejvm.com

Values,
Variables,
Types

Takeaways



immutable = can't be reassigned

```
var x: Int = 1
x = 1
x += 1
```

mutable

- prefer vals over vars
- all vals and vars have types
- compiler automatically infers types when omitted
- basic types
  - Boolean
  - Int, Long, Double
  - String

Expressions

Takeaways

```
val x = 3 + 5
val xIsEven = x % 2 == 0
val xIsOdd = !xIsEven
```

Basic expressions: operators

```
val cond: Boolean = ...
val i = if (cond) 42 else 0
```

if-structures are expressions

```
val x = {
  val cond: Boolean = ...
  if (cond) 42 else 0
}
```

code blocks are expressions

the value of the block is the value of its last expression

Expressions vs. instructions

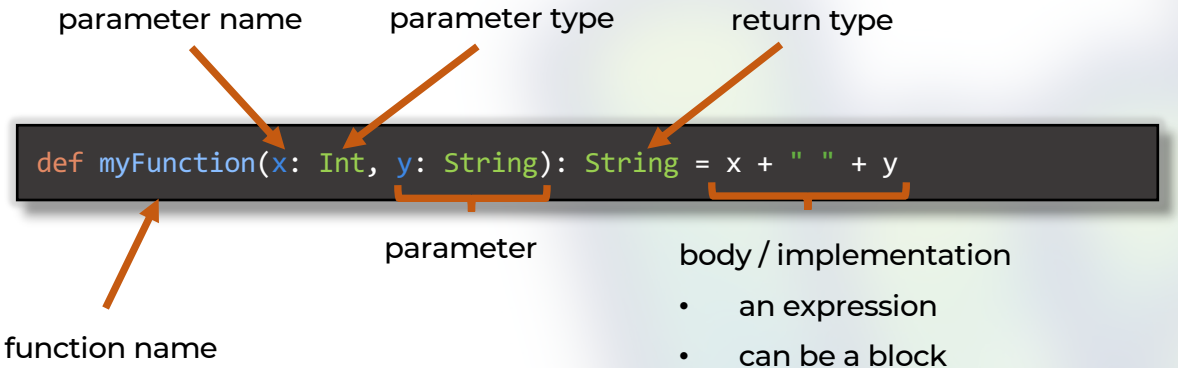
- instructions are executed (think Java), expressions are evaluated (Scala)
- in Scala we'll think in terms of expressions

Do NOT use while loops in your Scala code or I'll haunt you.

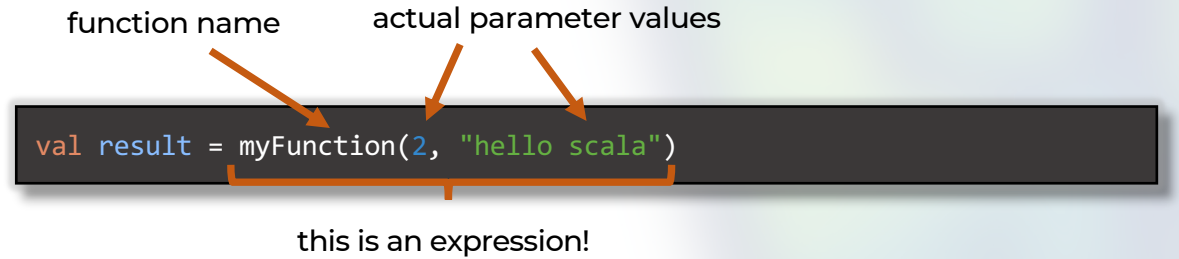
Functions

Functions - Takeaways

Defining functions



Calling functions



Functions - Takeaways

The Unit type

- used for side effects
- instruction-like functions

```
def aVoidFunction(x: Int): Unit
```

```
aVoidFunction(2)
println("hello world")
```

prints to console

Nested functions

- in blocks of code
- scoping

Basic recursion

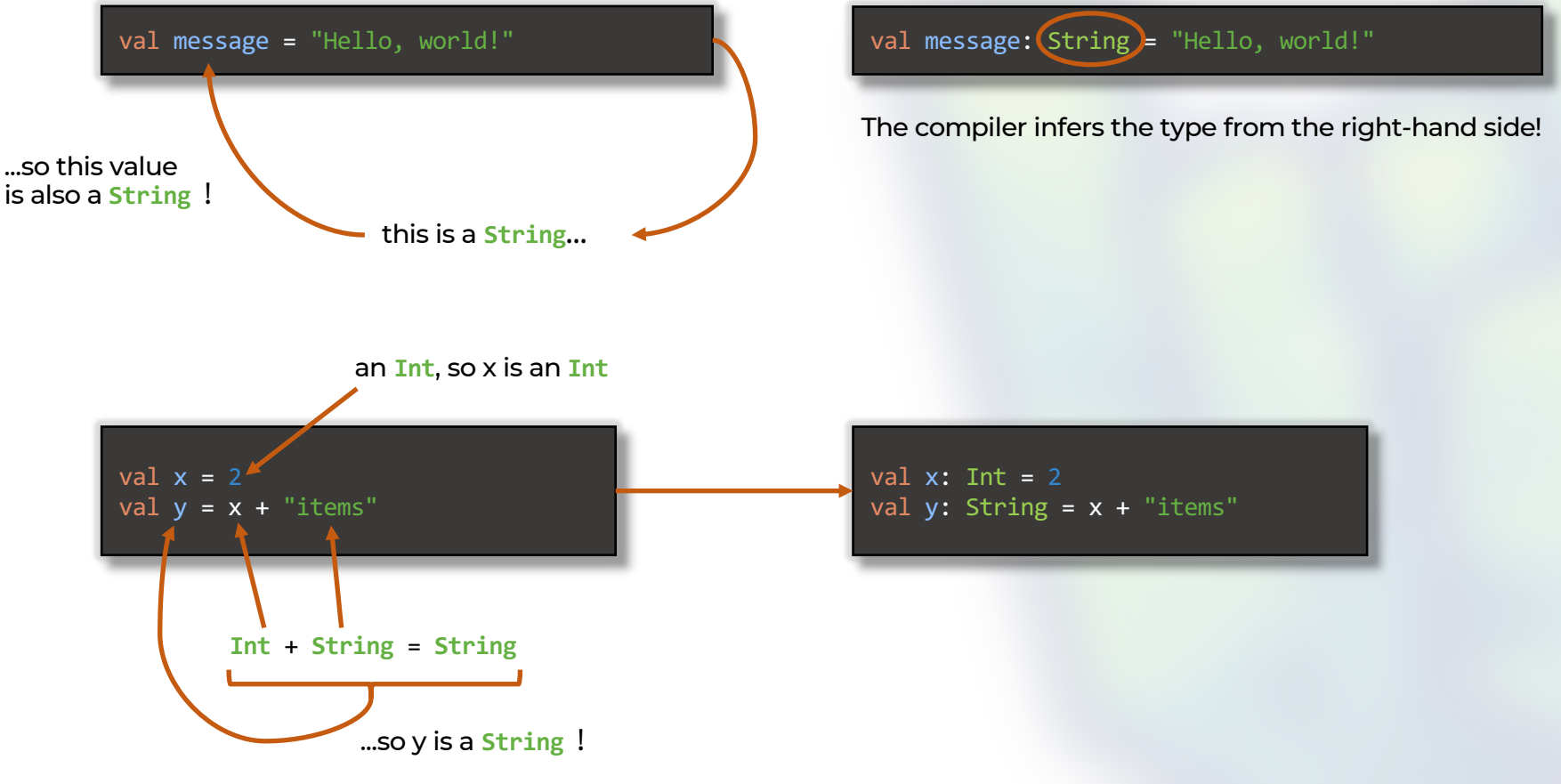
```
def isPrime(n: Int): Boolean = {
  def isPrimeAux(t: Int): Boolean =
    if (t <= 1) true
    else (n % t != 0) && isPrimeAux(t - 1)

  isPrimeAux(n / 2)
}
```

In Scala, we don't think in terms of loops, we think in terms of *recursion*.

Type Inference

What the Compiler Knows



What the Compiler *Also* Knows

```
def succ(x: Int) = x + 1
```

this is an `Int`...

...so the return type is `Int` !

```
def succ(x: Int): Int
```

The compiler infers the return type from the implementation

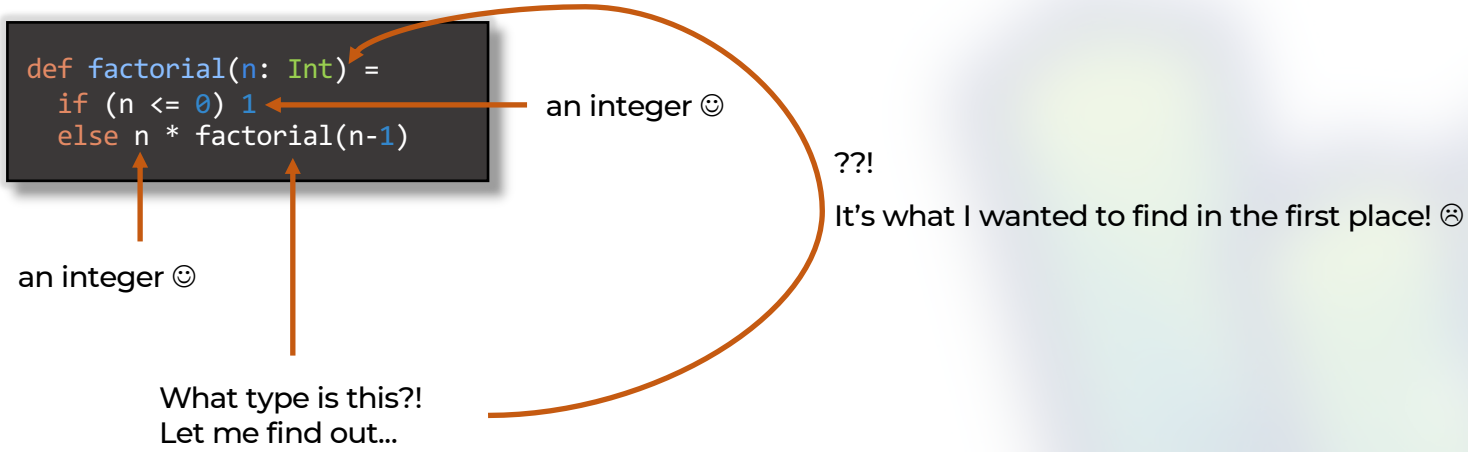
But don't try to fool it:

```
val x: Int = "Hello, world!"
```

an `Int` a `String`

?! ← type mismatch

Help the Compiler



Takeaway: for recursive functions, always specify the return type!

(as best practice: always specify the return type regardless)

Stack & Tail Recursion

Stack Recursion

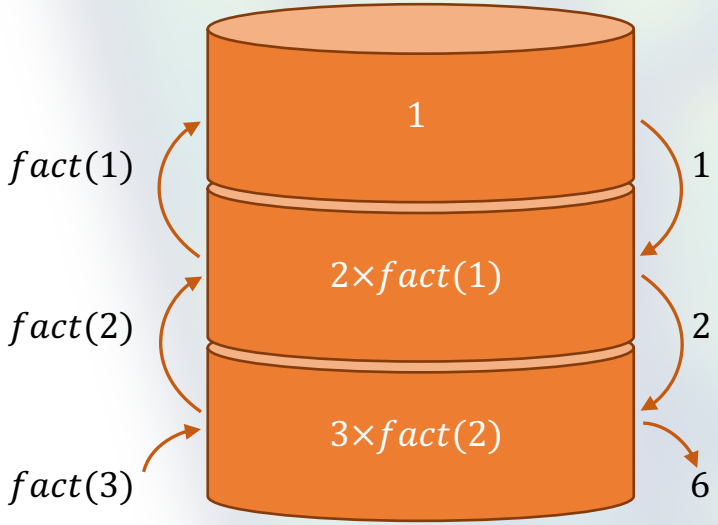
```
def factorial(x: Int): Int = {
  if (x <= 1) 1
  else x * factorial(x - 1)
}
```

we need to know the result of factorial(x - 1) to compute factorial(x)

the JVM must keep track of the call stack

The stack is limited

Can crash with stack overflow errors



Tail Recursion

- The recursive call occurs as the last call of any code path
- The JVM computes everything in the same stack frame

```
@tailrec def factorialHelper(x: Int, acc: Int): Int = {
  if (x <= 1) acc
  else factorialHelper(x - 1, x * acc)
}
```

if the method is not tail recursive we get a compile error:

Recursive call not in tail position

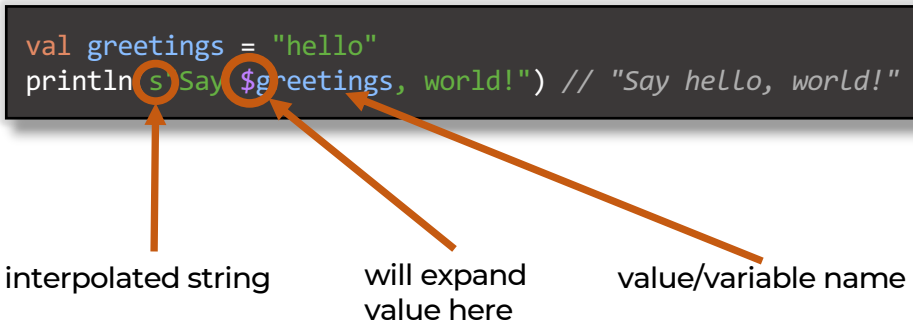
an accumulator keeps track of partial results

Other Basics

S-String Interpolators

Expands variables inside strings:

```
val greetings = "hello"
println(s"Say $greetings, world!") // "Say hello, world!"
```



Can also evaluate results:

```
val x = 2
val y = 3
println(s"$x + $y = ${x + y}") // "2 + 3 = 5"
```

Scala expression inside

Also F-interpolators and raw-interpolators (and some magical ones...)

Default Arguments

When 99% of time we call a function with the same params:

```
def factorial(x: Int, acc: Int = 1): Int = {
  ...
}
```

```
val fact10 = factorial(10)
```

the other argument is automatically inserted

Naming parameters

```
def greet(name: String = "Superman" age: Int = 10): String =
  s"Hi, I'm $name and I'm $age years old."
```

```
greet(age = 5)
greet(name = "Sally", age = 5)
greet(age = 5, name = "dog")
```

OO Basics

Takeaways

Defining classes

```
class Person(name: String, age: Int)
```

Instantiating

```
val bob = new Person("Bob", 25)
```

Parameters vs fields

```
class Person(val name: String, age: Int)
```

Defining methods

```
def greet(): String = { ... }
```

Calling methods

```
val bobSaysHi = bob.greet
```

- syntax allowed for parameter-less methods

The keyword `this`

Method Notations

```
class Person(name: String) {
  def likes(movie: String): Boolean = { ... }
  def unary_!(): String = { ... }
  def isAlive(): Boolean = { ... }
  def apply(greeting: String): String = { ... }
}
```

```
mary.apply("Hi there!")
mary("Hi there!")
```

apply() is special

```
mary.likes("Inception")
mary likes "Inception"
```

object method parameter

infix notation

- for methods with one parameter

```
mary.unary_!
!mary
```

prefix notation

- no parameters
- allowed for + - ! ~

```
mary.isAlive
mary isAlive
```

postfix notation

- no parameters
- deprecated since 2.13
- discouraged anyway

Objects

Takeaways

Scala doesn't have "static" values/methods

Scala objects

- are in their own class
- are the only instance
- implement the singleton pattern in one line!
- should not be confused with class instances

```
object Person {
  val N_EYES = 2
  def canFly: Boolean = false
}
```

Scala companions

- can access each other's private members
- Scala is more OO than Java!

```
class Person
object Person
```

Scala applications

```
def main(args: Array[String]): Unit
```

```
object MyApp extends App
```

Inheritance

Takeaways

Scala offers class-based inheritance

- access modifiers: `private`, `protected`, default (none = public)
- need to pass in constructor arguments to parent class

Derived classes can `override` members or methods

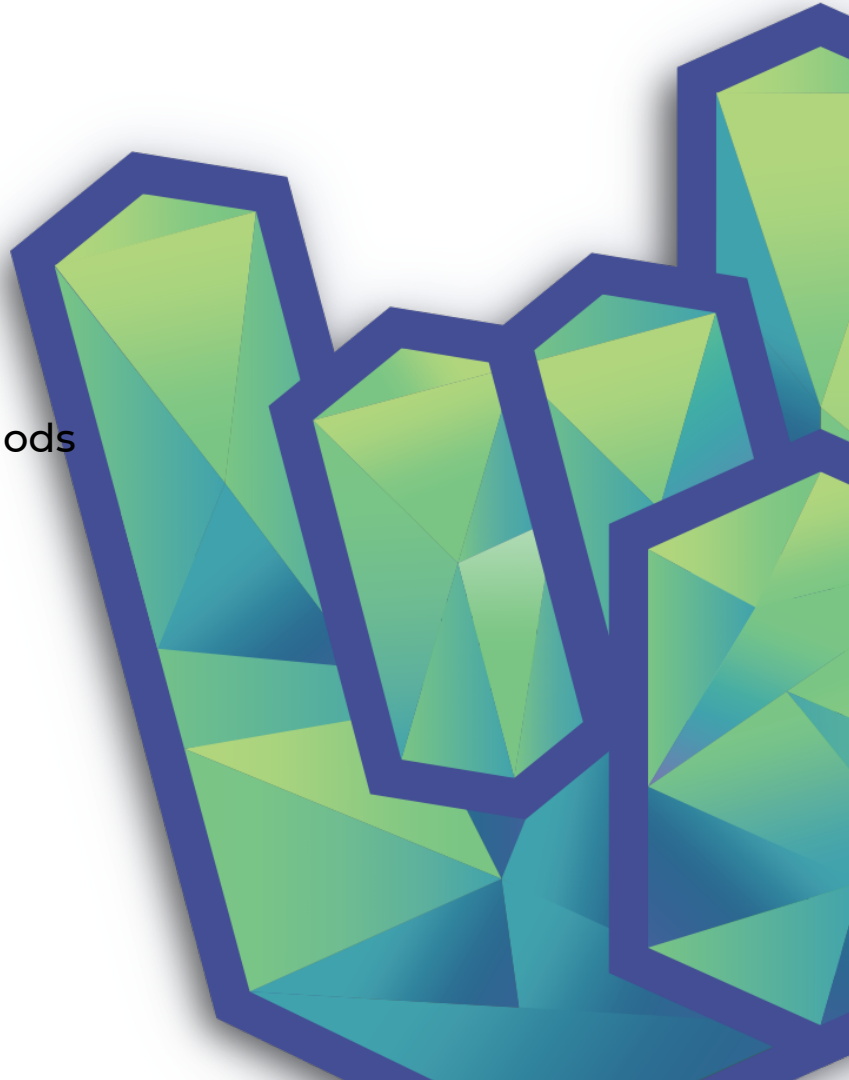
Reuse parent fields/methods with `super`

Prevent inheritance with `final` and `sealed`

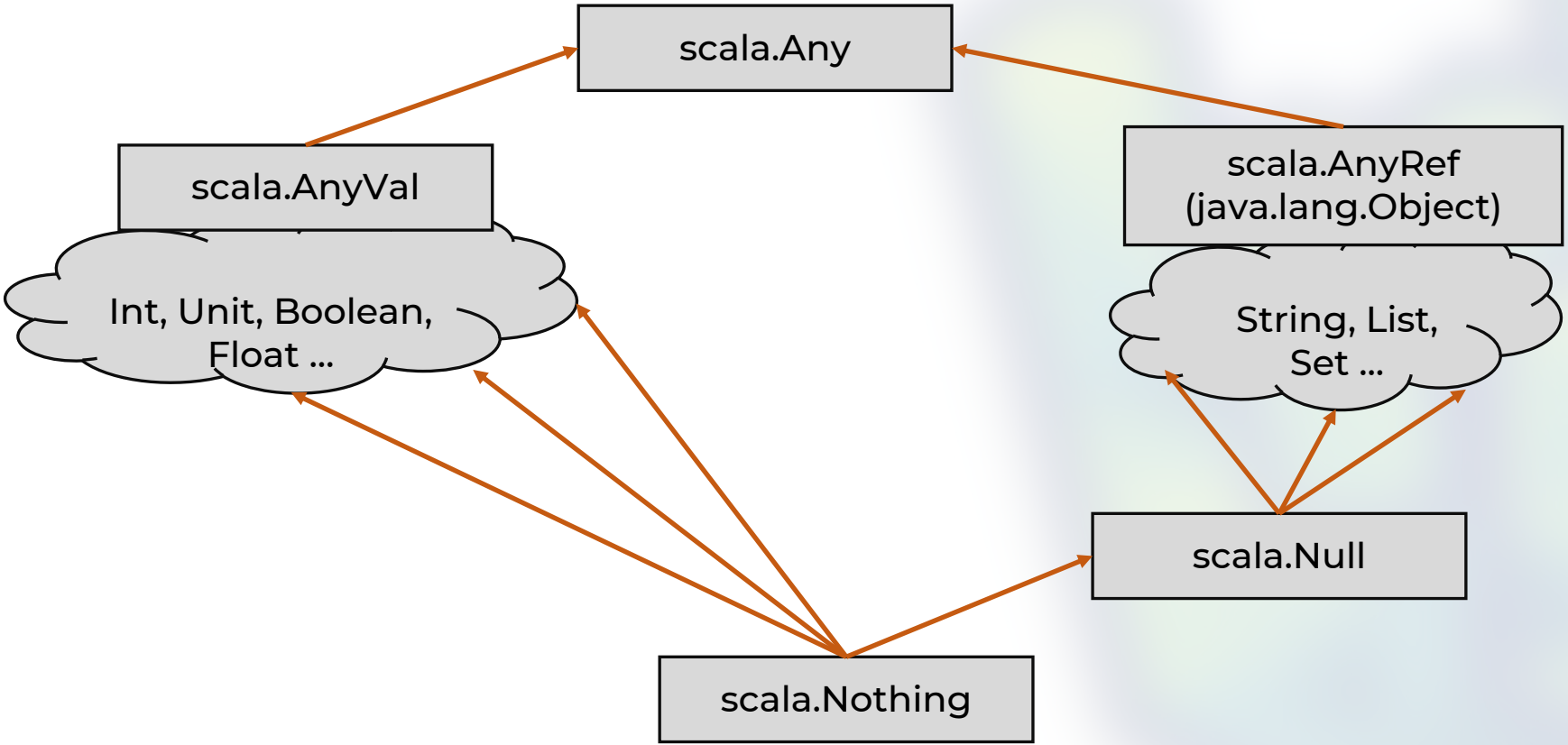
`abstract` classes

`trait`s

Inheriting from a class and multiple traits



Scala Type Hierarchy



Case Classes

Takeaways

Quick lightweight data structures with little boilerplate

```
case class Person(name: String, age: Int)
```

Companions already implemented

```
val bob = Person("Bob", 26)
```

Sensible equals, hashCode, toString

Auto-promoted params to fields

```
bob.name
```

Cloning

case objects

Generics

Takeaways

Use the same code on many (potentially unrelated) types:

```
trait List[T] {
  def add(elem: T)
}
```

Generic methods

```
object List {
  def single[A](element: A): List[A] = ???
}
```

Multiple type parameters

```
trait Map[Key, Value] {
  ...
}
```

What's a Function, Really?

Takeaways

We want to work with functions:

- pass functions as parameters
- use functions as values

Problem: Scala works on top of the JVM

- designed for Java
- first-class elements: objects (instances of classes)

Solution: ALL Scala functions are objects!

- function traits, up to 22 params
- syntactic sugar function types

```
trait Function1[-A, +B] {
  def apply(element: A): B
}
```

```
Function2[Int, String, Int]
```

```
(Int, String) => Int
```

Anonymous Functions

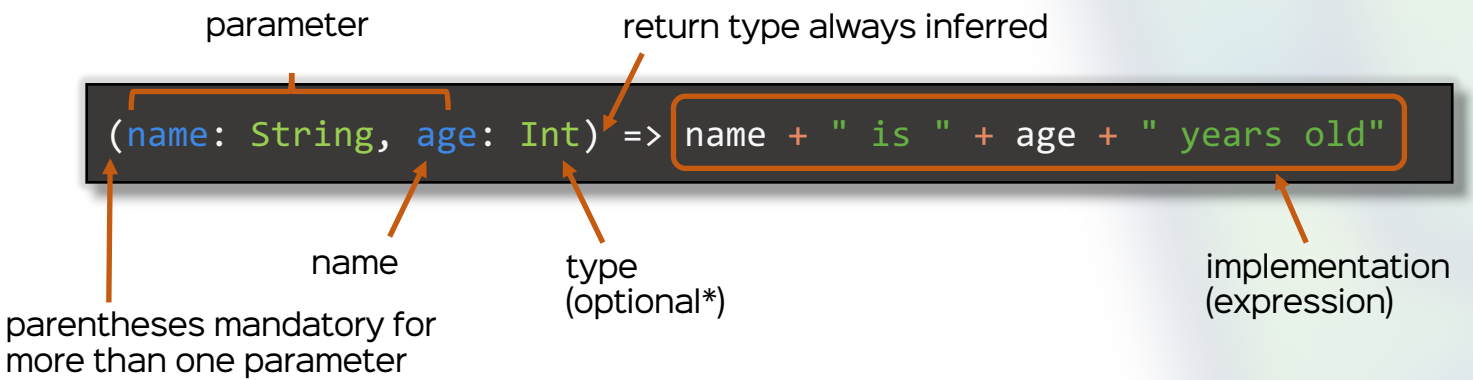
Takeaways

Instead of passing anonymous FunctionX instances every time

- cumbersome
- still object-oriented!

```
(x, y) => x + y
```

(lambda)



Moar sugar:

```
val add: (Int, Int) => Int = _ + _
```

HOFs & Curries

Takeaways

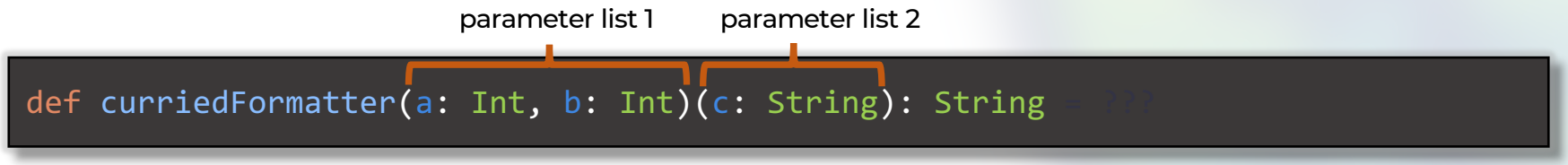
Functional programming = working with functions

- pass functions as parameters
- return functions as results

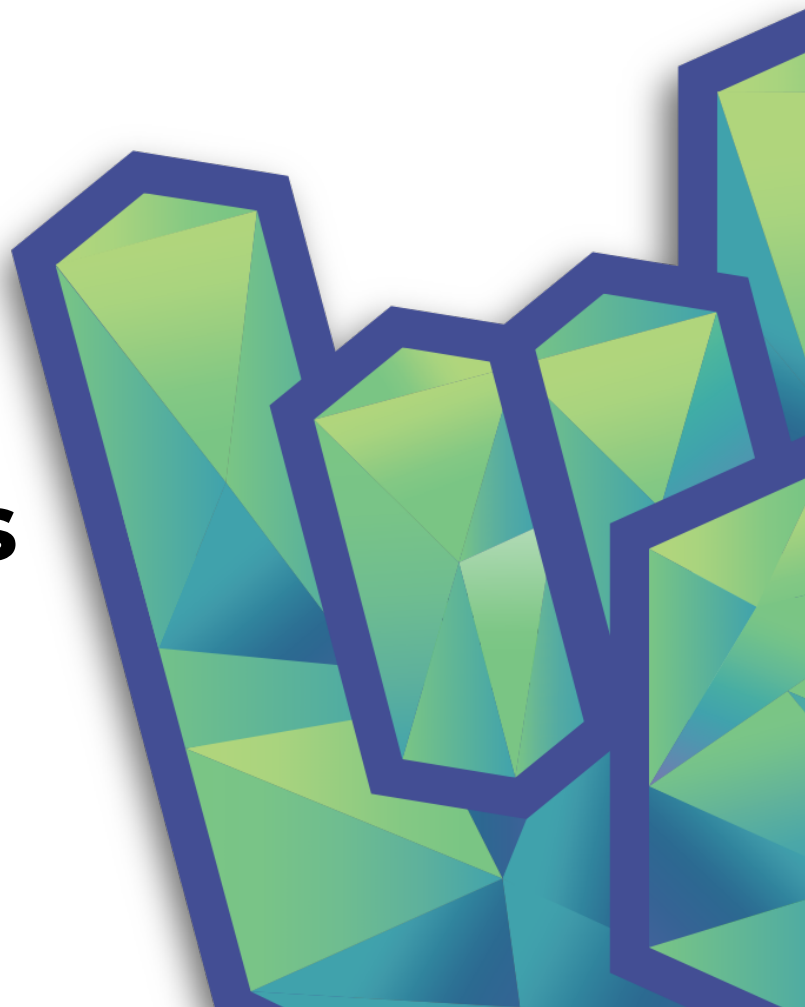
=> Higher Order Functions (HOFs)

```
def nTimesBetter(f: Int => Int, n: Int): Int => Int = ...
```

Currying = functions with multiple parameter lists



map, flatMap, filter, for-comprehensions



Takeaways

```
for {
  x <- List(1, 2, 3) if x % 2 != 0
  y <- List(4, 5, 6)
} yield x * y
```

```
List(1, 2, 3)
  .withFilter(_ % 2 != 0)
  .flatMap { x =>
    List(4, 5, 6).map(y => x * y)
  }
```

For comprehensions are rewritten into map/flatMap/filter calls

- multiple generators into flatMaps
- final yield into a map
- if-guards into withFilter (a specialized lazy* filter)

Use for comprehensions on complex chained calls for readability.

A Collections Overview

Sequences:

List, Array, Vector

Sequences

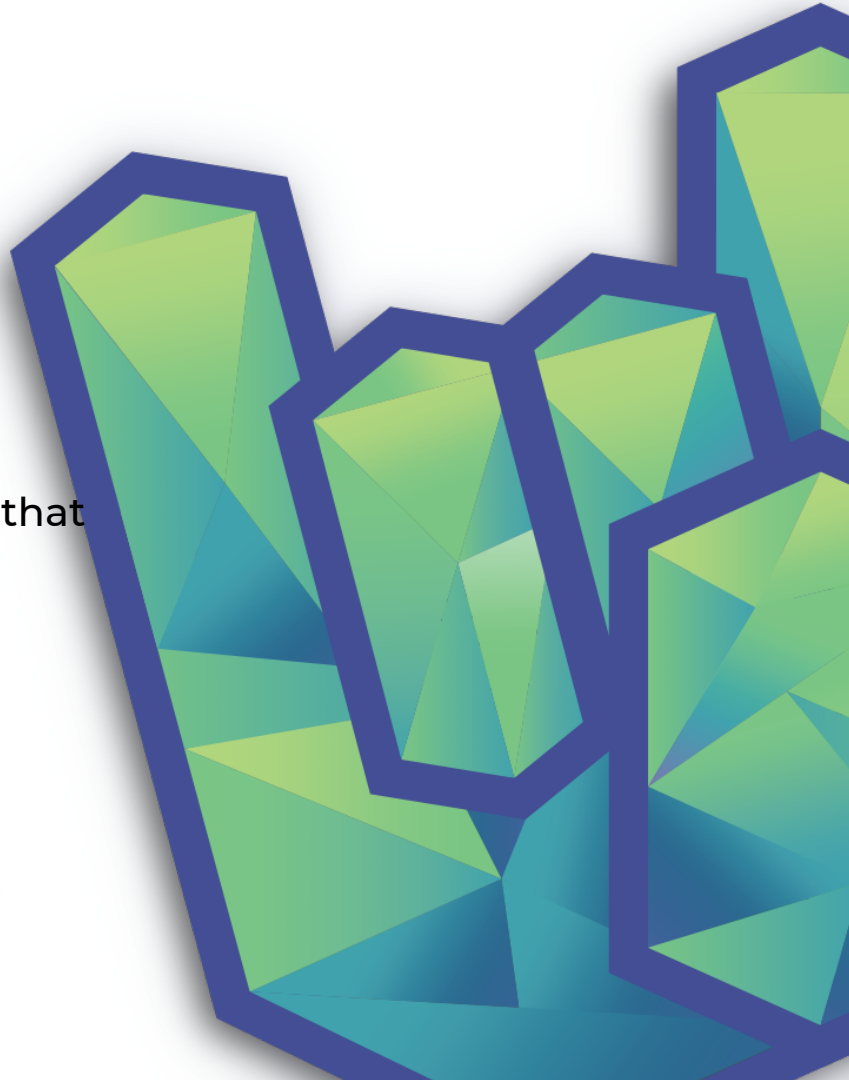
```
trait Seq[+A] {
  def head: A
  def tail: Seq[A]
}
```

A (very) general interface for data structures that

- have a well-defined order
- can be indexed

Supports various operations:

- apply, iterator, length, reverse for indexing and iterating
- concatenation, appending, prepending
- a lot of others: grouping, sorting, zipping, searching, slicing



List

```
sealed abstract class List[+A]
case object Nil extends List[Nothing]
case class ::[A](val hd: A, val tl: List[A]) extends List[A]
```

A LinearSeq immutable linked list

- head, tail, isEmpty methods are fast: O(1)
- most operations are O(n): length, reverse

Sealed - has two subtypes:

- object Nil (empty)
- class ::

Lists in Action

Vararg apply factory method for building instances

```
val numbers: List[Int] = List(1,2,3)
```

Some of the most used operations: accessors and transforms

```
numbers.head                          // 1
numbers.tail                          // List(2,3)
numbers map (_ + 1)                   // List(2,3,4)
```

Handy for comprehensions

```
for (x <- numbers if x > 2)
  yield (x, x * x)                    // List((3,9))
```

Lists in Action (2)

Lists reuse references to tails

sugar for
`::.apply(42, simple)`

```
// structural re-use
val simple = List(54)
val prepended = 42 :: simple

prepended.tail == simple            // true
```

A lot of useful functions are supported out of the box

```
// other utilities
List.fill(3)("apples")              // List("apples","apples","apples")
numbers foreach println             // prints all elements to console
numbers mkString "; "               // 1; 2; 3
numbers.reverse                     // List(3,2,1)
```

Arrays

```
final class Array[T]
    extends java.io.Serializable
    with java.lang.Cloneable
```

The equivalent of simple Java arrays

- can be manually constructed with predefined lengths
- can be mutated (updated in place)
- are interoperable with Java's T[] arrays
- indexing is fast
- offer the same Seq operators

Where's the Seq in the definition?!*

Arrays in Action

Instantiating "manually", with an exact length

```
val allocatedArray = new Array[String](3)   // filled with nulls
```

Instantiating via the apply vararg factory method

```
val numbers = Array(1,2,3,4)
```

Multi-dimensional arrays = arrays of arrays

```
val multiDimArray = Array.ofDim(2,3)        // a 2 x 3 array
```

Accessing elements: use apply

```
numbers(3) // 4
```

Arrays in Action (2)

Modifying an element in-place: use update

```
// syntax sugar - rewritten as numbers.update(1, 9)
numbers(1) = 9
```

Standard functions, uniform with the other Seqs

```
numbers :+ 5          // Array(1,9,3,4,5)
numbers.reverse       // Array(4,3,9,1) via a LOT of magic
```

Automatic conversion to Seq

```
// implicit* conversion to Seq[Int]
val numbersSeq: Seq[Int] = numbers
// utilities
numbersSeq.sum              // 17
numbersSeq.toList           // List(1,9,3,4)
```

also available in all collections:

toArray, toMap, toSet, toSeq, toStream*

Vector

```
final class Vector[+A]
```

The default implementation for *immutable sequences*

- *effectively constant* indexed read and write: $O(\log_{32}(n))$
- fast element addition: append/prepend
- implemented as a fixed-branched trie (branch factor 32)
- good performance for large sizes

```
val noElements = Vector.empty
val numbers = noElements :+ 1 :+ 2 :+ 3     // Vector(1,2,3)
val modified = numbers updated (0, 7)       // Vector(7,2,3)
```

Tuples & Maps

Takeaways

Tuples

```
val tuple = (42, "RockTheJVM")
tuple._1            // 42
tuple.copy(_1 = 0)    // (0,RockTheJVM)
tuple.toString        // "(42,RockTheJVM)"
tuple.swap            // (RockTheJVM,42)
```

retrieve elements using _n

create new tuples

pretty print

swap elements

Maps

```
val phonebook = Map("Jim" -> 555, "Mary" -> 789)
phonebook.contains("Jim")
val anotherbook = phonebook + ("Daniel", 123)
```

Functionals:

- filterKeys, mapValues
- map, filter, flatMap (on pairings!)

To/from other collections

- .toList, .toMap
- groupBy

Scala Rocks