Scala's static type system

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

More answers to "Why use Scala?"

> Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

More answers to "Why use Scala?"

> Scala combines object-oriented and functional programming in one concise, high-level language. **Scala's static types help avoid bugs in complex applications**, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

Some definitions

- **Type**: restricts the possible values to which a variable can refer, or an expression can produce, at run time

Scala value types have equivalent Java types

Scala types

- `scala.Double`
- `scala.Float`
- `scala.Long`
- `scala.Int`
- `scala.Short`
- `scala.Byte`
- `scala.Char`
- `scala.Boolean`
- `scala.Unit`

Java types

- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Long`
- `java.lang.Integer`
- `java.lang.Short`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Boolean`

Some definitions

- **Type**: restricts the possible values to which a variable can refer, or an expression can produce, at run time
- **Compile time**: when source code is translated into machine code, i.e., code that a computer can read
- **Run time**: when the program is executing commands (after compilation, if compiled)

Type systems

Static type systems

A language is statically typed if the type of a variable is known at compile time. That is, types checked before run-time.

- C/C++
- Fortran
- Java
- Scala

Dynamic type systems

A language is dynamically typed if types are checked on the fly. That is, types are checked during execution (i.e., run time).

- JavaScript
- Python
- Ruby
- R

Pros of static type systems

- Increased performance at run time
- Properties of your program verified (i.e., prove the absence of common type-related bugs)
- Safe refactorings
- Documentation in the form of type annotations (`: Int` in `val fourHearts: Int = 4`)

Cons of static type systems

- It takes time to check types (i.e., delay before execution)
- Code is verbose (i.e., code is longer/more annoying to write)
- The language is not flexible (e.g., one strict way of composing a type)

Reducing verbosity (with variables)

Without type inference

```
scala> val fourHearts: Int = 4
```

```
fourHearts: Int = 4
```

With type inference

```
scala> val fourHearts = 4
```

```
fourHearts: Int = 4
```

Reducing verbosity (with collections)

Without type inference

```
scala> val players: Array[String] = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

With type inference

```
scala> val players = Array("Alex", "Chen", "Marta")
```

```
players: Array[String] = Array(Alex, Chen, Marta)
```

Promoting flexibility

- Pattern matching
- Innovative ways to write and compose types

Compiled, statically-typed languages

Compiled languages

- Increased performance at run time

Statically-typed languages

- Increased performance at run time

Let's practice!

INTRODUCTION TO SCALA

Make decisions with if and else

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

A program for playing Twenty-One

Variables

```
val fourHearts: Int = 4
var aceClubs: Int = 1
```

Collections

```
val hands: Array[Int] = new Array[Int](3)`
```

Functions

```
// Define a function to determine if hand busts
def bust(hand: Int) = {
  hand > 21
}
```

Control structures

> A control structure is a block of programming that analyses variables and chooses a direction in which to go based on given parameters. The term flow control details the direction the program takes (which way program control "flows").

- `if` / `else`

[1] https://en.wikiversity.org/wiki/Control_structures

A single if

```
// This hand's point value
val hand = 24


// If this hand busts, print to output
if (hand > 21) {
  println("This hand busts!")
}
```

```
This hand busts!
```

A single if

```
// This hand's point value
val hand = 18

// If this hand busts, print to output
if (hand > 21) {
  println("This hand busts!")
}
```

if-else control flow

```
def maxHand(handA: Int, handB: Int): Int = {
  if (handA > handB) handA
  else handB
}
```

if-else control flow

```
// Point values for two competing hands
val handA = 17
val handB = 19

// Print the value of the hand with the most points
if (handA > handB) println(handA)
else println(handB)
```

if-else control flow

```
// Point values for two competing hands
val handA = 17
val handB = 19

// Print the value of the hand with the most points
if (handA > handB) {
  println(handA)
}
else {
  println(handB)
}
```

if-else control flow

```
// Point values for two competing hands
val handA = 17
val handB = 19

// Print the value of the hand with the most points
if (handA > handB)
  println(handA)
else
  println(handB)
```

if-else control flow

```
// Point values for two competing hands
val handA = 17
val handB = 19

// Print the value of the hand with the most points
if (handA > handB) println(handA)
else println(handB)
```

```
19
```

if-else control flow

```
// Point values for two competing hands
val handA = 17
val handB = 19

// Print the value of the hand with the most points
if (handA > handB) println(handA) else println(handB)
```

```
19
```

if-else if-else

```
// Point values for two competing hands
val handA = 26
val handB = 20


// If both hands bust, neither wins
if (bust(handA) & bust(handB)) println(0)
// If hand A busts, hand B wins
else if (bust(handA)) println(handB)
// If hand B busts, hand A wins
else if (bust(handB)) println(handA)
// If hand A is greater than hand B, hand A wins
else if (handA > handB) println(handA)
// Hand B wins otherwise
else println(handB)
```

if-else if-else

```
// Point values for two competing hands
val handA = 26
val handB = 20

// Find and print the best hand
if (bust(handA) & bust(handB)) println(0)
else if (bust(handA)) println(handB)
else if (bust(handB)) println(handA)
else if (handA > handB) println(handA)
else println(handB)
```

```
20
```

if expressions result in a value

```
scala> val handA = 17
```

```
handA: Int = 17
```

```
scala> val handB = 19
```

```
handB: Int = 19
```

```
scala> val maxHand = if (handA > handB) handA else handB
```

```
maxHand: Int = 19
```

Relational and logical operators

Relational

Greater than: `>`

Less than: `<`

Greater than or equal to: `>=`

Less than or equal to: `<=`

Equal to: `==`

Not equal to: `!=`

Logical

And: `&&`

Or: `||`

Not: `!`

These "operators" are actually methods!

Relational

Greater than: `>`

Less than: `<`

Greater than or equal to: `>=`

Less than or equal to: `<=`

Equal to: `==`

Not equal to: `!=`

Logical

And: `&&`

Or: `||`

Not: `!`

These "operators" are actually methods!

Relational

Greater than: `>`

Less than: `<`

Greater than or equal to: `>=`

Less than or equal to: `<=`

Equal to: `==`

Not equal to: `!=`

Logical

And: `&&`

Or: `||`

Not: `!`

[1] "Future music" is an oft [2] used phrase within DataCamp. It means "something you want or aspire to have but not for the near future" and is of Dutch origin.

Let's practice!

INTRODUCTION TO SCALA

while and the imperative style

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

Control structures

- `if` `else`
- `while`

Hip hip hooray

Dancing smiley by Krabat der Zauberlehrling

[1] https://en.wikipedia.org/wiki/Hip_hip_hooray

Loop with while

```
// Define counter variable
var i = 0


// Define the number of times for the cheer to repeat
val numRepetitions = 3


// Loop to repeat the cheer
while (i < numRepetitions) {
  // BODY OF LOOP
}
```

Loop with while

```
// Define counter variable
var i = 0

// Define the number of times for the cheer to repeat
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i = i + 1
}
```

Loop with while

```
// Define counter variable
var i = 0

// Define the number of times for the cheer to repeat
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i += 1 // i = i + 1
}
```

Loop with while

```
// Define counter variable
var i = 0

// Define the number of times for the cheer to repeat
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i += 1 // ++i and i++ don't work!
}
```

i = 0

```
// Define variables for while loop
var i = 0
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i = i + 1
}
```

```
Hip hip hooray!
```

i = 1

```
// Define variables for while loop
var i = 0
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i = i + 1
}
```

```
Hip hip hooray!
Hip hip hooray!
```

i = 2

```
// Define variables for while loop
var i = 0
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i = i + 1
}
```

```
Hip hip hooray!
Hip hip hooray!
Hip hip hooray!
```

i = 3

```
// Define variables for while loop
var i = 0
val numRepetitions = 3

// Loop to repeat the cheer
while (i < numRepetitions) {
  println("Hip hip hooray!")
  i = i + 1
}
```

```
Hip hip hooray!
Hip hip hooray!
Hip hip hooray!
```

Loop with while over a collection

```
// Define counter variable
var i = 0


// Create an array with each player's hand
var hands = Array(17, 24, 21)


// Loop through hands and see if each busts
while (i < hands.length) {
  // BODY OF LOOP
}
```

Scala is object-oriented

- Rule of thumb: pretty much everything is an object in Scala

```
scala> var hands = Array(17, 24, 21)
scala> hands.length
```

```
res0: Int = 3
```

Loop with while over a collection

```
// Define counter variable
var i = 0

// Create an array with each player's hand
var hands = Array(17, 24, 21)

// Loop through hands and see if each busts
while (i < hands.length) {
  println(bust(hands(i)))
  i = i + 1
}
```

Loop with while over a collection

```
var i = 0
var hands = Array(17, 24, 21)
while (i < hands.length) {
  println(bust(hands(i)))
  i += 1
}
```

```
false
true
false
```

Like if, parentheses required for while

The `while` loop:

```
while (i < hands.length)
```

The `maxHand` function:

```
if (handA > handB)
```

Like if, parentheses required for while

The `while` loop:

```
while i < hands.length
```

The `maxHand` function:

```
if handA > handB
```

The imperative style

```
var i = 0
var hands = Array(17, 24, 21)
while (i < hands.length) {
  println(bust(hands(i)))
  i += 1
}
```

```
false
true
false
```

Let's practice!

INTRODUCTION TO SCALA

foreach and the functional style

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

Scala is functional

Scala is functional

1. Functions are first-class values
2. Operations of a program should map input values to output values rather than change data in place

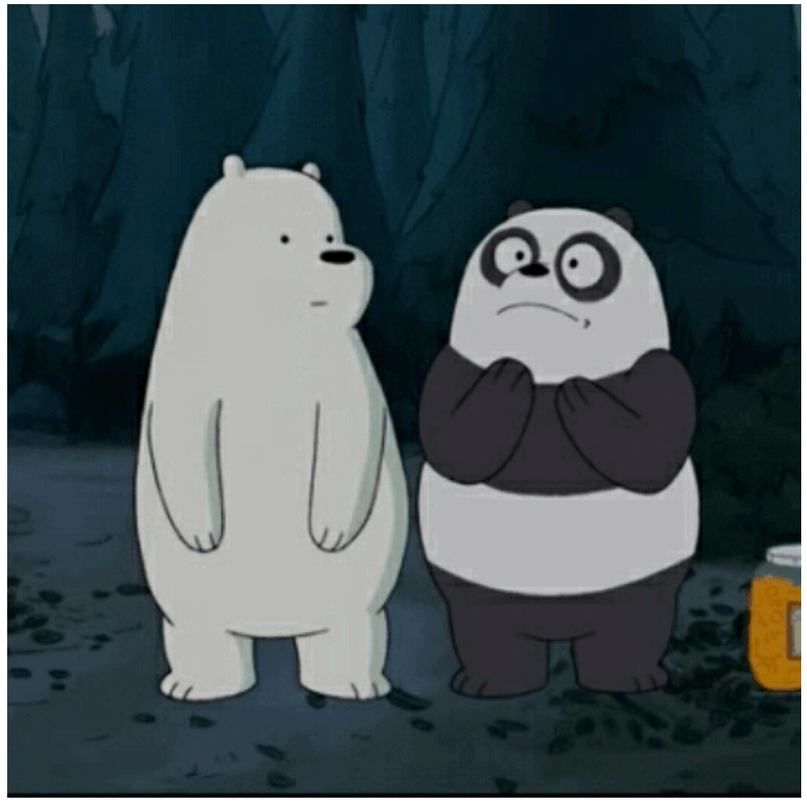
Scala is a imperative/functional hybrid

Scala *usually* is functional but can also be imperative sometimes

Scala is functional:

1. Functions are first-class values
2. Operations of a program should map input values to output values rather than change data in place

Scala nudges us towards the functional style



functional

"imperative" according to Oxford Dictionary

Imperative (English):

- Definition: Of the nature of or expressing a command.

[1] https://www.thefreedictionary.com/imperative

The imperative style

Scala *usually* is functional but can also be imperative sometimes

Scala can be imperative:

- One command at a time
- Iterate with loops
- Mutate shared state (e.g., mutating variables out of scope)
- Examples: C, Java, Python

The imperative style

```
// Define counter variable
var i = 0

// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// Loop through hands and see if each busts
while (i < hands.length) {
  println(bust(hands(i)))
  i = i + 1
}
```

The imperative style

```
// Define counter variable
var i = 0

// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// Loop through hands and see if each busts
while (i < hands.length) {
  println(bust(hands(i)))
  i = i + 1
}
```

[1] http://bit.ly/state_wikipedia

The functional style

Scala *usually* is functional but can also be imperative sometimes

Scala is functional:

1. Functions are first-class values

The functional style

- Functions are first-class values

```
// Define counter variable
var i = 0

// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// Loop through hands and see if each busts
while(i < hands.length) {
  println(bust(hands(i)))
  i += 1
}
```

The functional style

- Functions are first-class values

```
// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// See if each hand busts
hands.foreach(INSERT FUNCTION HERE)
```

Scala fuses OOP and FP -> Scala is scalable

- Functions are first-class values

```
// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// See if each hand busts
hands.foreach(INSERT FUNCTION HERE)
```

From Chapter 1:

> Scala combines object-oriented and functional programming

Modify the bust function

```
// Define a function to determine if hand busts
def bust(hand: Int) = {
  println(hand > 21)
}
```

The functional style

- Functions are first-class values

```
// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// See if each hand busts
hands.foreach(INSERT FUNCTION HERE)
```

The functional style

- Functions are first-class values

```
// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// See if each hand busts
hands.foreach(bust)
```

```
false
true
false
```

What is a side effect?

Scala *usually* is functional but can also be imperative sometimes

Scala is functional:

1. Functions are first-class values
2. Operations of a program should map input values to output values rather than change data in place

Side effect: *code that modifies some variable outside of its local scope*

[1] http://bit.ly/side_effect_wikipedia

What is a side effect?

```
def bust(hand: Int) = {
  println(hand > 21)
}
```

```
scala> val myHand = bust(22)
```

```
true
myHand: Unit = ()
```

What is a side effect?

```
// Define counter variable
var i = 0


// Initialize array with each player's hand
var hands = Array(17, 24, 21)


// Loop through hands and see if each busts
while (i < hands.length) {
  println(bust(hands(i)))
  i = i + 1
}
```

The spectrum of functional style

Less functional than before

```
// Define a function to determine if hand busts
def bust(hand: Int) = {
  println(hand > 21)
}
```

More functional than before

```
// Initialize array with each player's hand
var hands = Array(17, 24, 21)

// See if each hand busts
hands.foreach(bust)
```

[1] http://bit.ly/array_foreach_documentation

Signs of style

Imperative

- `var`
- Side effects
- `Unit`

Functional

- `val`
- No side effects
- Non- `Unit` value types
  - `Int`
  - `Boolean`
  - `Double`

[1] http://bit.ly/scala_unit_documentation

Let's practice!

INTRODUCTION TO SCALA

The essence of Scala

INTRODUCTION TO SCALA

David Venturi
Curriculum Manager, DataCamp

The final video!

[1] Dancing smiley by Krabat der Zauberlehrling

The functional style

- Operations of a program should map input values to output values rather than change data in place

Non-functional code

- **WHEN** operations of a program **DON'T** map input values to output values and **DO** change data in place

The functional style

- Operations of a program should map input values to output values rather than change data in place

Benefits of the functional style

- Your data won't be changed inadvertently
- Your code is easier to reason about
- You have to write fewer tests
- Functions are more reliable and reusable

Scala is a hybrid imperative/functional language

Prefer

- `val`
- Immutable objects
- Functions without side effects

If necessary

- `var`
- Mutable objects
- Functions with side effects

What's next

Programming

- Functions
- More collections
- More types
- Object-oriented programming
- Functional programming
- Pattern matching
- Concurrency
- More...

Data

- Scala for the data engineer
- Scala for the data scientist
- Scala for the machine learning engineer

The essence of Scala

[1] This course is in beta so the program that plays the game is not included in this video. Check back in a week or so and it will be! :)

The essence of Scala

> Fusing functional and object-oriented programming

[1] http://bit.ly/scala_days_odersky_2018

The essence of Scala

> Fusing functional and object-oriented programming **in a statically-typed setting**

[1] http://bit.ly/scala_days_odersky_2018

Congratulations!

INTRODUCTION TO SCALA