Scala Course

Basics 1

47 Deg

2021-01-29

Scala Basics

Scala?

- Object Oriented/Functional Language

Scala Basics

Scala?

- Object Oriented/Functional Language
- Static typing

Scala Basics

Scala?

- Object Oriented/Functional Language
- Static typing
- Type inference

Scala Basics

Scala?

- Object Oriented/Functional Language
- Static typing
- Type inference
- Functional programming capabilities

Scala Basics

Why object oriented?

- Subtyping polymorphism

Scala Basics

Why object oriented?

- Subtyping polymorphism
- Everything's an object

Scala Basics

Why object oriented?

- Subtyping polymorphism
- Everything's an object
- Inheritance

Scala Basics

Why functional?

- Functions are *first class citizens*

Why functional?

- Functions are *first class citizens*
- Pattern matching

Scala Basics

Why functional?

- Functions are *first class citizens*
- Pattern matching
- Algebraic Data Types (ADT)

Scala Basics

Why functional?

- Functions are *first class citizens*
- Pattern matching
- Algebraic Data Types (ADT)
- Higher kinded types

Why functional?

- Functions are *first class citizens*
- Pattern matching
- Algebraic Data Types (ADT)
- Higher kinded types
- Immutability

Scala Basics

Why functional?

- Functions are *first class citizens*
- Pattern matching
- Algebraic Data Types (ADT)
- Higher kinded types
- Immutability

Scala Basics

Why functional?

- Functions are *first class citizens*
- Pattern matching
- Algebraic Data Types (ADT)
- Higher kinded types
- Immutability

This is the cool part, and what we'll be learning about

The type system

```scala
val a: Int = "hola!"
// error: type mismatch;
//  found   : String("hola!")
//  required: Int
```

A type system checks the types of our program to be sure they make sense.

Static typing

Static typing means that once a variable has a type, it cannot change. The opposite of static typing is dynamic typing, as in Python, JS...

Python

```python
a = "patata"
a = 3
```

Static typing

Static typing means that once a variable has a type, it cannot change. The opposite of static typing is dynamic typing, as in Python, JS...

JS

```
var a = "patata"
a = 3
```

Static typing

Static typing means that once a variable has a type, it cannot change. The opposite of static typing is dynamic typing, as in Python, JS...

Scala

```scala
var a = "patata"
a = 3
// error: type mismatch;
//  found   : Int(3)
//  required: String
// val times3: Int => Int =
//
```

Type inference

Normally, the compiler will try to guess what type our values have even if we don't specify it.

```scala
val hola = "hola"
val three = 3
```

This doesn't mean that if we don't specify a type the declaration remains untyped, just that we let the compiler guesses it.

Higher order functions

Functions being first class citizens means that they can be used as any other value in your program. They have types, they can be returned, and they can be taken as parameters.

Functions are values

```scala
val intToString: Int => String = { a =>
  a.toString
}
```

Higher order functions

Functions being first class citizens means that they can be used as any other value in your program. They have types, they can be returned, and they can be taken as parameters.

```
def applyFunction(
  number: Int,
  fn: Int => Int
): Int =
  fn(number)

applyFunction(3, {x => x*3})
applyFunction(2, {x => x+2})
```

Higher order functions

Functions being first class citizens means that they can be used as any other value in your program. They have types, they can be returned, and they can be taken as parameters.

```scala
def genMultiplication(
  times: Int
): Int => Int = { x =>
  x * times
}

val times3: Int => Int = genMultiplication(3)
times3(3)
```

Generics

Generics are a vital part of abstraction in functional programming. It allows us to parametrize functions, classes, traits to make them work for arbitrary types. Let's see an example:

Generics

Generics are a vital part of abstraction in functional programming. It allows us to parametrize functions, classes, traits to make them work for arbitrary types. Let's see an example:

```
def compose(
  f: String => Int,
  g: Int => Boolean
): String => Boolean = { str =>
  g(f(str))
}
```

Generics

Generics are a vital part of abstraction in functional programming. It allows us to parametrize functions, classes, traits to make them work for arbitrary types. Let's see an example:

```
val length: String => Int = _.length
val isEven: Int => Boolean = { i => i % 2 == 0 }

val lengthIsEven = compose(length, isEven)

lengthIsEven("1")
lengthIsEven("10")
lengthIsEven("100")
lengthIsEven("1000")
```

Generics

Although, in the previous example, do we really care about the specific types of the functions `f` & `g`? or we just need them to match?

We could use a **generic** implementation for `compose`!

```
def composeGeneric[A, B, C](
  f: A => B,
  g: B => C
): A => C = { a =>
  g(f(a))
}
```

Generics

And then, we could use `composeGeneric` the same way we used `compose`.

```scala
val length: String => Int = _.length
val isEven: Int => Boolean = { i => i % 2 == 0 }

val lengthIsEven = composeGeneric(length, isEven)

lengthIsEven("1")
lengthIsEven("10")
lengthIsEven("100")
lengthIsEven("1000")
```

Algebraic data types

Algebraic data types are composite types made up from smaller ones. There are two basic constructs for them:

- case classes
- sealed traits

Case classes

case classes, also called **product types**, encode a grouping of other fields that should **all** be present in all values.

```
case class Package(
  length: Int,
  width: Int,
  height: Int,
  weight: Int)
```

In this example we know that all packages should have a length, a width, a height, and a weight.

Case classes

Case classes have other cool feature, copying. Copying allows us to create copies of the object with some fields changed. This helps to make programs immutable!

```scala
// Notice that, for instantiating case classes,
// we don't use `new`.
val pack = Package(10, 15, 20, 3)
```

Case classes

If we want to change one of the fields of a case class, we just need to call copy and reassign a new value for the field:

```scala
val pack2 = pack.copy(weight = 2)
```

Sealed traits

Sealed traits *(**sealed** means we can only extend it on the file we've declared)*, also called **sum types**, encode a type that can be **one of** all the different invariants:

```
sealed trait ResponseFromServer
case class OkResponse(
  json: String
) extends ResponseFromServer
case object FailureResponse extends ResponseFromServer
```

Here, we know that a value of the type `ResponseFromServer` can be either a `OkResponse` or a `FailureResponse`.
*(**sealed** means we can only extend it on the file we've declared)*

Recap.

Scala	Other languages
`trait`	`interface`(Java), `protocol` (Swift)
`class`	`class`
`case class`	`POJO/JavaBean` (Java)
`object`	class containing only static stuff
`val`	immutable reference (`const` in JS)
`var`	mutable reference
`def`	function/method

Exercise 1.1

Let's imagine a simple event based application. We want to define some events that we can handle:

- A user logs in
- A customer adds an item to the basket
- A user starts payment process
- Payment goes through correctly
- Payment process fails with timeout
- Payment process fails because of Insufficient funds

solution

One possible solution... not **the** one.

```
import java.util.UUID

sealed trait Event
case class UserLogIn(userId: UUID) extends Event
case class AddItemToBasket(userId: UUID, itemId: UUID) ex
case class UserIntentPay(userId: UUID) extends Event
case class PaymentCorrect(userId: UUID, paymentReceipt: S

sealed trait PaymentFailure extends Event
case class TimeoutFailure(userId: UUID, intentId: UUID) e
case class InsufficientFundsFailure(userId: UUID, intentI
```

Exercise 1.2

We know that all events for this system will have several fields: - Event ID - User ID

Refactor your previous exercise to add those.

solution

```
sealed trait Event {
  def id: UUID
  def userId: UUID
}

case class UserLogIn(id: UUID, userId: UUID) extends Even
case class AddItemToBasket(id: UUID, userId: UUID, itemId
case class UserIntentPay(id: UUID, userId: UUID) extends
case class PaymentCorrect(id: UUID, userId: UUID, payment

sealed trait PaymentFailure extends Event
case class TimeoutFailure(id: UUID, userId: UUID, intentI
case class InsufficientFundsFailure(id: UUID, userId: UUI
```