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 (wiki)

Scala Basics

Scala?

- Object Oriented/Functional Language
- Static typing (wiki)
- Type inference

Scala Basics

Scala?

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

Scala Basics

Why object oriented?

- Subtyping polymorphism (wiki)

Scala Basics

Why object oriented?

- Subtyping polymorphism (wiki)
- Everything's an object

Scala Basics

Why object oriented?

- Subtyping polymorphism (wiki)
- Everything's an object
- Inheritance

Why functional?

- Functions are *first class citizens*

Scala Basics

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

Scala Basics

Why functional?

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

Object Oriented vs FP equivalent

pattern/principle

- SRP

pattern/principle

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
• SRP • Open/Closed Principle	

Object Oriented vs FP equivalent

pattern/principle

- SRP
- Open/Closed Principle
- Dependency Inversion

pattern/principle

Object Oriented vs FP equivalent

pattern/principle

- SRP
- Open/Closed Principle
- Dependency Inversion
- Interface Segregation

pattern/principle

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	
Open/Closed Principle	
Dependency Inversion	
Interface Segregation	
Factory pattern	

Object Oriented vs FP equivalent

pattern/principle

- SRP
- Open/Closed Principle
- Dependency Inversion
- Interface Segregation
- Factory pattern
- Strategy pattern

pattern/principle

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
• SRP • Open/Closed Principle • Dependency Inversion • Interface Segregation • Factory pattern • Strategy pattern • Decorator pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
• SRP • Open/Closed Principle • Dependency Inversion • Interface Segregation • Factory pattern • Strategy pattern • Decorator pattern • Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	
Dependency Inversion	
Interface Segregation	
Factory pattern	
Strategy pattern	
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	
Interface Segregation	
Factory pattern	
Strategy pattern	
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	
Factory pattern	
Strategy pattern	
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	Functions
Factory pattern	
Strategy pattern	
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	Functions
Factory pattern	Functions
Strategy pattern	
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	Functions
Factory pattern	Functions
Strategy pattern	Functions
Decorator pattern	
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	Functions
Factory pattern	Functions
Strategy pattern	Functions
Decorator pattern	Functions
Visitor pattern	

Object Oriented vs FP equivalent

pattern/principle	pattern/principle
SRP	Functions
Open/Closed Principle	Functions
Dependency Inversion	Functions
Interface Segregation	Functions
Factory pattern	Functions
Strategy pattern	Functions
Decorator pattern	Functions
Visitor pattern	Functions

Object Oriented vs FP equivalent

The problem is Functional patterns are different

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
// applyFunction(3, {x => x*
//                  ^
```

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.

Syntax

Values - Immutable

```scala
val immutable: Int = 1
immutable = 2
// error: reassignment to val
// val intToString: Function[Int, String] =
//                                         ^
```

Syntax

Variables - Mutable

```scala
var mutable: Int = 1
// mutable: Int = 1
mutable = 2
mutable
// res3: Int = 2
```

Syntax

Methods

```scala
def identity(i: Int): Int = i
identity(1)
// res4: Int = 1
```

- Blocks aren't needed for one-lines
- `return` isn't needed: last statement is returned

Syntax

Call by value

We already know this. When we pass a parameter to a method "normally"

```scala
def callByValue(i: Int): Int = i
callByValue(1)
// res5: Int = 1
```

Syntax

Call by name

The parameter won't be evaluated until needed.

```scala
def callByName(i: => Int): Int = 1
def inf: Int = inf
callByName(inf)
// res6: Int = 1
```

Syntax

Call by name

The parameter won't be evaluated until needed.

```scala
def callByName(i: => Int): Int = i
def inf: Int = inf
callByName(inf)
// error: ambiguous reference to overloaded definition,
// both method inf in object App of type Int
// and  method inf in object App of type Int
// match expected type Int
// def inf: Int = inf
//                ^^^
// error: ambiguous reference to overloaded definition,
// both method inf in object App of type Int
// and  method inf in object App of type Int
// match expected type Int
```

Syntax

Classes

```scala
class MyInt(val i: Int) {
  def sum(j: Int): Int = i + j
}
val myInt = new MyInt(1)
// myInt: MyInt = repl.MdocSession$App$MyInt@681969a4
myInt.i
// res8: Int = 1
myInt.sum(1)
// res9: Int = 2
```

- free constructor
- `val` means `i` is public (otherwise it'd be private)

Syntax

Classes

```
class MyIntPrivate private(val i: Int) {
  def sum(j: Int): Int = i + j
}
new MyIntPrivate(1)
// error: constructor MyIntPrivate in class MyIntPrivate
// new MyIntPrivate(1)
//     ^^^^^^^^^^^^^
```

- this is how make the constructor private

Syntax

Objects

```scala
object MyObject {
  def sum(i: Int, j: Int): Int = i + j
}
MyObject.sum(1,1)
// res11: Int = 2
```

- Objects are single instances of their own definitions

Syntax

Traits

```scala
trait MyTrait {
  val i: Int
  def sum(j: Int): Int = i + j
}
new MyTrait {
 val i: Int = 1
}.sum(1)
// res12: Int = 2
```

- They are like Java interfaces
- You can have default implementations

Syntax

Traits

```scala
class MyClass(j: Int) extends MyTrait {
  val i = j
}
new MyClass(1).sum(1)
// res13: Int = 2
```

Syntax

type

```scala
type Money = Int

def gimmeMoney(n: Money): Unit = println(n)

val treeEur: Money = 3
// treeEur: Money = 3

gimmeMoney(3)
// 3
```

Syntax

Although, you can use also the *typed type*

type

```scala
type Money = Int

def gimmeMoney(n: Money): Unit = println(n)

val treeEur: Int = 3
// treeEur: Int = 3

gimmeMoney(3)
// 3
```

Syntax

This is the basic. You can find more:
https://docs.scala-lang.org/tour/basics.html

Scala Type Hierarchy

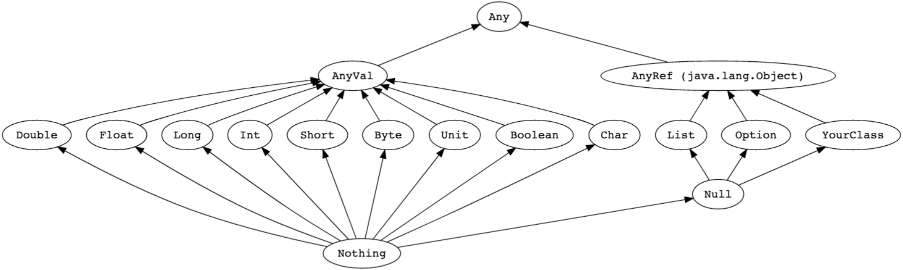

Figure 1: scala type hierarchy

- `Any` is the supertype of all types, also called the top type.
- `AnyVal` represents value types: `Int`, `Double`, etc.
- `AnyRef` represents reference types: e.g. `String` (everything else)
- `Null` is a subtype of all reference types (type, not `null` value)
- `Nothing` is a subtype of all types, also called the bottom type.

The type system

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

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

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
}
intToString(1)
```

Higher order functions

Actually, this is possible because functions are traits

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

This is the same as `A => B`. It's syntactic sugar.

Higher order functions

```scala
val intToString: Function[Int, String] =
  new Function[Int, String] {
    def apply(a: Int): String = a.toString
  }
// intToString: Int => String = <function1>
intToString(1)
// res20: String = "1"
```

Higher order functions

```scala
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: Int => Int = <function1>
times3(3)
// res23: Int = 9
```

Higher order functions

Currification

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

val times3: Int => Int = genMultiplication(3)
// times3: Int => Int = <function1>
times3(3)
// res25: Int = 9
```

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:

```scala
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:

```scala
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))
}
```

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.

```scala
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
```