Scala Object System

Mikhail Mutcianko, Alexey Otts

СПБгУ, СП

27 февраля 2020 г.

Recap: Syntax Differences from Kotlin

Syntax

Kotlin ⇔ Scala

Syntax

`Kotlin` ⇔ `Scala`
`List<Integer>` ⇔ `List[Int]`

Syntax

`Kotlin`	⇔	`Scala`
`List<Integer>`	⇔	`List[Int]`
`fun foo(x: Integer): A { }`	⇔	`def foo(x: Int): A = { }`

Syntax

Kotlin	⇔	Scala
`List<Integer>`	⇔	`List[Int]`
`fun foo(x: Integer): A { }`	⇔	`def foo(x: Int): A = { }`
`import foo.*`	⇔	`import foo._`

Syntax

`Kotlin`	⇔	`Scala`
`List<Integer>`	⇔	`List[Int]`
`fun foo(x: Integer): A { }`	⇔	`def foo(x: Int): A = { }`
`import foo.*`	⇔	`import foo._`
`when(x) { }`	⇔	`x match { }`

Syntax

Kotlin	⇔	Scala
`List<Integer>`	⇔	`List[Int]`
`fun foo(x: Integer): A { }`	⇔	`def foo(x: Int): A = { }`
`import foo.*`	⇔	`import foo._`
`when(x) { }`	⇔	`x match { }`
`class Foo : Bar`	⇔	`class Foo extends Bar`

Syntax

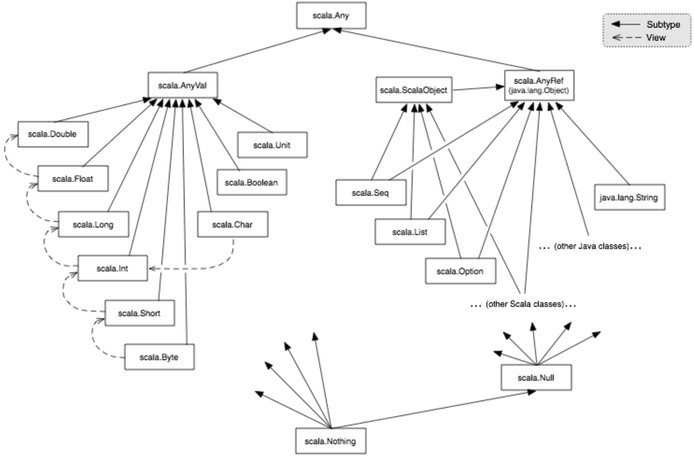
Kotlin	⇔	Scala
`List<Integer>`	⇔	`List[Int]`
`fun foo(x: Integer): A { }`	⇔	`def foo(x: Int): A = { }`
`import foo.*`	⇔	`import foo._`
`when(x) { }`	⇔	`x match { }`
`class Foo : Bar`	⇔	`class Foo extends Bar`
`->`	⇔	`=>`

Syntax

Kotlin	⇔	Scala
List<Integer>	⇔	List[Int]
fun foo(x: Integer): A { }	⇔	def foo(x: Int): A = { }
import foo.*	⇔	import foo._
when(x) { }	⇔	x match { }
class Foo : Bar	⇔	class Foo extends Bar
->	⇔	=>
it	⇔	_

Scala Type System

Everything is a type



Top types

- `Any`
  - The base type of all types
  - Methods: `==`, `!=`, `equals`, `hashCode`, `toString`

Top types

- `Any`
  - The base type of all types
  - Methods: `==`, `!=`, `equals`, `hashCode`, `toString`
- `AnyRef`
  - The base type of all reference types
  - Alias of `java.lang.Object`

Top types

- `Any`
  - The base type of all types
  - Methods: `==`, `!=`, `equals`, `hashCode`, `toString`
- `AnyRef`
  - The base type of all reference types
  - Alias of `java.lang.Object`
- `AnyVal`
  - The base type of all primitive types

Bottom type

Nothing is at the bottom of Scala's type hierarchy. It is a subtype of every other type.

There is no value of type Nothing.

Why is that useful?

- To signal abnormal termination
- As an element type of empty collections

Classes, Objects, Traits . . .

Basic

```scala
class Useless

class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}
```

Class constructor

- In Scala, a class implicitly introduces a primary constructor
- Takes the parameters of the class
- Executes all statements in the class body
- Can introduce members if parameters are specified as `val` or `var`

Class constructor

Example

```
class Calculator(brand: String) {
  /*  A constructor */
  val colour: String = if (brand == "TI") {
    "blue"
  } else if (brand == "HP") {
    "black"
  } else {
    "white"
  }
  def add(m: Int, n: Int): Int = m + n
}
```

Auxiliary Constructors

Scala also allows the declaration of auxiliary constructors.

These are methods named `this`

```scala
class Calculator(brand: String) {
  val color: String = ???

  def this(other: Cal) = ??? // <<<

  // An instance method.
  def +(m: Int, n: Int): Int = m + n
}
```

Objects

- Scala has no static classes / methods / fields
- Objects are used to hold single instances of a class
- Objects are lazy, and are not initialized until first reference

Objects

Example

```scala
object Constants {
  val e = 2.71828182846
  def **(num: Double) = num * e
}

Constants ** Constants.e
```

Companion objects

A companion object in Scala is an object that's declared in the same file as a class, and has the same name as the class

```scala
class SomeClass {
    def printFilename() = {
        println(SomeClass.HiddenFilename)
    }
}

object SomeClass {
    private val HiddenFilename = "/tmp/foo.bar"
}
```

Traits

Traits are collections of fields and behaviors that you can extend or mixin to your classes

- In Scala, a class can only have one superclass, but many traits
- Can have methods and properties
- Can have member definitions
- Cannot have constructors

Traits

Example

```
trait Planar {
  def height: Int
  def width: Int
  def surface = height * width // <<<
}

class Square extends Shape with Planar with Movable
```

Abstarct classes

- Can have methods and properties
- Can have member definitions
- Can have constructors

Super calls

Problem

To keep your Scala code DRY (“Don’t Repeat Yourself ”), you want to invoke a method that’s already defined in a parent class or trait.

Super calls

Example

```
class WelcomeActivity extends Activity {
    override def onCreate(bundle: Bundle) {
        super.onCreate(bundle)
        // more code here ...
    }
}
```

Super calls

Controlling which trait you call a method from

```
class Child extends Human with Mother with Father {
    def printSuper = super.hello
    def printMother = super[Mother].hello
    def printFather = super[Father].hello
    def printHuman = super[Human].hello
}
```

Members

- `def` - method
- `val` - immutable property
- `var` - mutable property
- `lazy val` - lazy immutable property

Properties

All properties implicitly generate a backing field, a setter and a getter

Except for `private` ones

Properties

```
class Person() {
 private var name = ""
 var age = 0
}
```

Properties

```
class Person() {
 private var name = ""
 var age = 0
}
```

```
class Person() {
 private var name = ""
 // Private age variable, renamed to _age
 private var _age = 0

 // Getter
 def age = _age

 // Setter
 def age_= (value:Int):Unit = _age = value
}
```

Concrete member overriding

- `val`: can only be overridden by val
- `lazy val`: can only be overridden by lazy val
- `var`: a concrete var cannot be overridden
- `def`: can be overridden by all kinds of members

	val	lazy	var	def
val	✓	✗	✗	✗
lazy	✓	✗	✗	✗
var	✗	✗	✗	✗
def	✓	✓	✓	✓

Abstract member overriding

- `lazy val`: cannot be abstract
- `val`: can be overridden by val and lazy val
- `var`: can be overridden by var, or a pair of read and write operations implemented by def, val, or lazy val
- `def`: can be overridden by all kinds of members

	val	var	def
val	✓	✗	✗
var	✓+	✓	✓+
def	✓	✓	✓

Acess modifiers

Problem

Scala methods are public by default, and you want to control their scope in ways similar to Java.

Acess modifiers

Problem

Scala methods are public by default, and you want to control their scope in ways similar to Java.

Scopes

- object-private scope
- private
- package
- protected

Acess modifiers

Problem

Scala methods are public by default, and you want to control their scope in ways similar to Java.

Scopes

- object-private scope
- private
- package
- protected

Access modifiers

object-private

The most restrictive access is to mark a method as “object-private.” When you do this, the method is available only to the current instance of the current object. Other instances of the same class cannot access the method.

```scala
class Foo {
    private[this] def isFoo = true
    def doFoo(other: Foo) {
        if (other.isFoo) { // this line won’t compile
            // ...
        }
    }
}
```

Access modifiers

package scope

```
package com.acme.coolapp.model {
    class Foo {
        private[model] def doX {}
        private def doY {}
    }
    class Bar {
        val f = new Foo
        f.doX // compiles
        f.doY // won’t compile
    }
}
```

apply method

apply is a syntactic sugar for calling something

- construct new instances
- directly call anonymous functions
- mimic callable behaviour
- see also: update method

apply method

example

```scala
class SomeClass(val x: Int)

object SomeClass {
  def apply(x: Int) = new SomeClass(x)
}

new SomeClass(42)
SomeClass(42)
```

Packages

- declare one or more package names at the top of a Scala file

```scala
package users
class User
```

Packages

- declare one or more package names at the top of a Scala file

```scala
package users
class User
```

- declare packages by using braces

```scala
package users {
  package administrators {
    class NormalUser
  }
  package normalusers {
    class NormalUser
  }
}
```

Imports

- single name

```
import users.User
```

Imports

- single name

```
import users.User
```

- all from package

```
import users._
```

Imports

- single name

```
import users.User
```

- all from package

```
import users._
```

- only import selected names

```
import users.{User, UserPreferences}
```

Imports

- single name

```scala
import users.User
```

- all from package

```scala
import users._
```

- only import selected names

```scala
import users.{User, UserPreferences}
```

- rename imported member

```scala
import users.{UserPreferences => UPrefs}
```

Package objects

Scala provides package objects as a convenient container shared across an entire package

Package objects

Scala provides package objects as a convenient container shared across an entire package

- can contain arbitrary definitions

Package objects

Scala provides package objects as a convenient container shared across an entire package

- can contain arbitrary definitions
- can inherit from other classes and traits

Package objects

Scala provides package objects as a convenient container shared across an entire package

- can contain arbitrary definitions
- can inherit from other classes and traits
- one package object per package

Package objects

Scala provides package objects as a convenient container shared across an entire package

- can contain arbitrary definitions
- can inherit from other classes and traits
- one package object per package
- should be placed in `package.scala`

Package obejcts

```scala
package com.myapp
package object model {
  val MAGIC_NUM = 42 // field
  def echo(a: Any) { println(a) } // method
}
```

Package obejcts

```scala
package com.myapp
package object model {
  val MAGIC_NUM = 42 // field
  def echo(a: Any) { println(a) } // method
}
```

```scala
package com.myapp.model
class MainDriver {
  echo(42)
  echo(MAGIC_NUM)
}
```

Functional OOP

Extractor object

An extractor object is an object with an unapply method.

`apply`

takes arguments and creates an object

`unapply`

takes an object and tries to give back the arguments

`unapply` method

Return type

- If it is just a test, return a `Boolean`
- If it returns a single sub-value of type `T`, return an `Option[T]`
- If you want to return several sub-values `T1,...,Tn`, group them in an optional tuple `Option[(T1,...,Tn)]`

Extractor object

Example

```scala
class Cat(val name: String, val age: Int)

object Cat {
  def apply(name: String, age: Int): Cat = new Cat(name, age)
  def unapply(cat: Cat): Option[(String, Int)] = Some(cat.name -> cat.age)
}
```

Sealed traits

- sealed traits can only be extended in the same file

Sealed traits

- sealed traits can only be extended in the same file
- this lets the compiler easily know all possible subtypes

Sealed traits

- sealed traits can only be extended in the same file
- this lets the compiler easily know all possible subtypes
- example: the compiler can emit warnings if a match/case expression is not exhaustive

Sealed traits

- sealed traits can only be extended in the same file
- this lets the compiler easily know all possible subtypes
- example: the compiler can emit warnings if a match/case expression is not exhaustive
- use sealed traits when the number of possibly subtypes is finite and known in advance

Sealed traits

- sealed traits can only be extended in the same file
- this lets the compiler easily know all possible subtypes
- example: the compiler can emit warnings if a match/case expression is not exhaustive
- use sealed traits when the number of possibly subtypes is finite and known in advance
- they can be a way of creating something like an enum in Java

Sealed traits

- sealed traits can only be extended in the same file
- this lets the compiler easily know all possible subtypes
- example: the compiler can emit warnings if a match/case expression is not exhaustive
- use sealed traits when the number of possibly subtypes is finite and known in advance
- they can be a way of creating something like an enum in Java
- they help you define algebraic data types, or ADTs

Sealed traits

Example

```
sealed trait Answer
case object Yes extends Answer
case object No extends Answer
case class Maybe(something: String)
```

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Case classes auto generate:

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Case classes auto generate:

- properties based on constructor arguments

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Case classes auto generate:

- properties based on constructor arguments
- `equals`, `canEqual`, `hashCode` and `toString` based on properties

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Case classes auto generate:

- properties based on constructor arguments
- `equals`, `canEqual`, `hashCode` and `toString` based on properties
- companion object with `apply` and `unapply` methods

Clase classes

`Case classes` are used to conveniently store and match on the contents of a class.

You can construct them without using new

Case classes auto generate:

- properties based on constructor arguments
- `equals`, `canEqual`, `hashCode` and `toString` based on properties
- companion object with `apply` and `unapply` methods
- `copy` method with default values

Destructuring bindings

Initializing a value may be more than binding a name — it’s a pattern.

Destructuring bindings

Initializing a value may be more than binding a name — it's a pattern.

```
val pi = 3.1415
val pi: Double = 3.1415 // equivalent to first definition
val c@Cat(name, age) = Cat("bobik", 42) // a pattern definition
val x :: xs = c :: name :: age :: Nil  // an infix pattern definition
```

Pattern matching

Values

```scala
val times = 1

times match {
  case 1 => "one"
  case 2 => "two"
  case _ => "some other number"
}
```

Pattern matching

Guards

```
val times = 1

times match {
  case i if i == 1 => "one"
  case i if i == 2 => "two"
  case _ => "some other number"
}
```

Pattern matching

Types

```scala
val animal = Cat("eve", 10)

val owner = animal match {
  case a: Cat if a.name == "eve" => "Bob"
  case a: Dog if a.name == "dis" => "Mary"
  case _ => "Unknown"
}
```

Pattern matching

Unapply

```scala
val animal = Cat("eve", 10)

val owner = animal match {
  case Cat("eve", _) => "Bob"
  case Dog("dis", _) => "Mary"
  case _ => "Unknown"
}
```

Pattern matching

Binding

```
val animal = Cat("eve", 10)

val fullName = animal match {
  case c@Cat("eve", _) => s"Bob’s ${c.name}"
  case d@Dog("dis", _) => s"Mary’s ${d.name}"
  case _ => "Unknown"
}
```

Practice

Functional List