Классы и объекты в Scala

I. Class

class

Ключевое слово class

В классе могут быть объявлены поля и методы

В общем, концепция классов схожа с классами в других языках (C++/C#/Java/etc.)

Инициализация полей происходит при создании объекта класса

Если в теле класса определён вызов функции (вне определения какой-либо другой функции), этот вызов будет производиться каждый раз при создании объекта класса

```
class Person {
  //init fields inside ctor
  val Name = "Ivan"
  var Surname = "Surname"
  private def fullName() =
                    s"$Name $Surname"
  def sayfullName() = print(fullName)

  println("still in the ctor; ")
}
val p = new Person
//Out: still in the ctor;
p.Surname = "Soldatov"  (): Unit
p.sayfullName  (): Unit
//Out: "Ivan Soldatov"
```

I. Class

case class

Ключевое слово case class

Должен содержать список полей

Можно провести аналогию между кейс-классами в Scala и структурами (structure) в C/C++

Объекты кейс-классов сравниваются по значению (не по ссылке)

Pattern-matching

Копирование с использованием именованных аргументов

```
case class Person(name: String,
                  surname: String = "") {
  private def fullName=s"$name $surname"
  def sayFullName() = println(fullName)
}
val p0 = Person("Ivan")
val p1 = Person("Ivan", "Soldatov")
println(p0 == p1) //Out: false
val p2 = Person("Ivan")
println(p2 == p0) //Out: true
for (p ← Seq(p0, p1, p2)) //Out: Soldatov
  p match {
    case Person(n, s) ⇒ println(s)
  }
val p3 = p1.copy(surname = "Ivanov")
p3.sayFullName //Out: Ivan Ivanov
```

I. Class

конструкторы

Для каждого класса Scala всегда определён один конструктор по умолчанию

Ключевое слово this и def this

Для каждого класса Scala может быть определено несколько вспомогательных конструкторов (auxiliary constructors)

```
case class Person(var name: String,
                  var surname: String) {
  println(s"Ctor person $name $surname")
  def this(fullName: String) {//"=" not required
    //MUST be used default ctor in the first line
    this("empty", "empty")
    val strs = fullName.split(" ")
    this.name = strs.head
    this.surname = strs.lastOption.getOrElse("")
  } }
val p0 = new Person("Innokenty", "Goldberg")
//Out: Ctor person Innokenty Goldberg
val p1 = new Person("Innokenty Goldberg")
//Out: Ctor person empty empty
println(s"${p1.name} ${p1.surname}")
//Out: Innokenty Goldberg
```

I. Class

список аргументов конструктора и поля класса

Список аргументов конструктора класса и поля класса не эквивалентны!

По умолчанию, для аргументов конструктора класса не определены методы получения и передачи значения

Чтобы методы получения значения аргументов класса были определены, необходимо явно указать «val». Аналогично для методов передачи значения, необходимо использовать ключевое слово «var»

```
class someClass(arg: String) {
  val field: String = "field"
}
val someObj = new someClass("arg")
println(someObj.field) //Out: field
println(someObj.arr) //Err: no getter
class anotherClass0(val arg: String)
class anotherClass1(var arg: String)
val aObj0 = new anotherClass0("arg0")
val aObj1 = new anotherClass1("arg1")
println(s"${aObj0.arg} ${aObj1.arg}")
//Out: arg0 arg1
```

I. Class

список аргументов конструктора и поля класса

Для кейс-классов методы получения значения определены по умолчанию

Методы же передачи значения будут определены лишь в случае явного использования «var». Однако, не стоит прибегать к такой порочной практике: для кейс классов определён метод «copy»!

```
case class someClass(arg: String) {
  val field: String = "field"
}
val someObj = someClass("arg")
println(someObj.field) //Out: field
println(someObj.arg) //Ok, Out: arg
someObj.arg = "bad practice" //Err!
case class badClass(var arg: String)
val badObj = badClass("arg")
badObj.arg="bad practice"//Not ok but works
val niceObj = someObj.copy(arg = "new arg")
```

I. Class

конструкторы; как мог бы выглядеть Scala класс в C#

В приведённом примере:

Приведена аналогия реализации одного и того же класса в Scala и C#

Если (val name: String) в Scala, то public string name; (вместо private)

Если (var name: String) в Scala, то public string name; (без readonly)

Если нет полей в Scala, то нет полей и в C#, а конструктор в C# без параметров

```
/*----    Scala    ----*/
class Greeter(name: String) {
  def sayHi() = println(s"Greetings, $name")
}
/*----      C#     ----*/
public class Greeter {
  private readonly string name;
  public Greeter(string name) {
    this.name = name;
  }
  public void sayHi() {
    Console.WriteLine("Greetings, " + name);
  }
}
```

II. Object

object

В Scala отсутствует ключевое слово static

object гарантированно создаётся лишь единожды

Конструирование object происходит при первом вызове его метода либо при первом присваивании его переменной

```
class Foo {
  static def m = {} //Err: static?
}

object math {
  println("math constructed")
  val PI = 3.14159265359
  var arg: Double = _
  def circle: Double = 2.0 * PI * arg
}
val m0 = math //Out: math constructed
val m1 = math //No new output
m0.arg = 4.0
println(m1.circle) //Out:25.13274122872
println(m0 == m1) //Out: true
```

II. Object

companion object, метод apply

companion object:
- object и class называются одинаково
- определены в одном файле

Класс может получить доступ к закрытым (private) полям и методам companion object

Метод apply:
-определяется в object
-синтаксис вызова подобен вызову конструктора
-может конструировать экземпляры класса-компаньона
-может возвращать любой тип

```
class Circle(r: Double){
  import Circle._
  def area = calculateArea(r)
}
object Circle{
  private val PI = 3.14159265359
  private def calculateArea(r: Double)=PI*r*r
  def apply(r: Double) = new Circle(r)
  def apply() = PI
}
val c0 = new Circle(5.0)
val c1 = Circle(5.0) //without new
println(c0 == c1)
println(c0.area == c1.area)
println(Circle()) //Out: 3.14159265359
```

II. Object

case class companion object

Для кейс-класса неявно определён companion object с методом «apply», позволяющим конструировать экземпляры класса
По этой причине при создании экземпляров кейс-класс не обязательно использовать «new»

Если же переопределить метод «apply» в object companion, то можно получить не особо ожидаемое поведение

```
case class intWithStr(i: Int, s: String)
object intWithStr {
  def apply(i: Int, s: String) =
                        print("NotExpected")
}
val obj0 = intWithStr(0, "Zero")
//Out: NotExpected

//Is type of obj0 and obj1 intWithStr?
val obj1: intWithStr = intWithStr(1, "One")
//Err! type mismatch! obj1 is Unit!

val obj2: intWithStr = new intWithStr(2, "Two")
//That`s ok
```

III. Модификаторы доступа

модификаторы доступа: protected, private

В Scala отсутствует ключевое слово «public»

По умолчанию, все поля и методы публичные

protected: доступ разрешён внутри самой сущности, а также внутри сущностей-наследников

private: доступ разрешён только внутри самой сущности

Модификаторы доступа могут применяться к полям и методам, а также к самим сущностям

```
class Clazz {
  def pubPing = privPing
  protected def protPing = privPing
  private def privPing = println("PING")
}
class Clazz1 extends Clazz {
    pubPing //Out: PING
    protPing //Out: PING
    privPing //Err!
}
val c = new Clazz
c.pubPing //Out: PING
c.protPing //Err!
c.privPing //Err!
```

III. Модификаторы доступа

модификаторы доступа: private[this]

В Scala для модификаторов доступа private и protected допускается указание области действия

Синтаксис:
private[xxx] или protected[xxx], где «xxx» - имя пакета, либо имя сущности, либо «this»

private[this] схож с private, однако, посредством private[this] можно ограничить область доступа для companion object и для экземпляров самой сущности, которые используются в теле сущности

```
class WithPrivateThis {
  private val priv = "priv"
  private[this] val privThis="privThis"
  def foo(selfObj: WithPrivateThis) = {
    println(selfObj.priv) //Out: priv
    println(selfObj.privThis) //Err!
  }
}
object WithPrivateThis {
  //Out: priv
  println((new WithPrivateThis).priv)
  //Err!
  println((new WithPrivateThis).privThis)
  (new WithPrivateThis).foo
}
val obj = WithPrivateThis
```

IV. Интерфейсы

trait

trait в Scala во многом подобен interface в C#/Java

trait может содержать объявления полей и методов, а также их определения

В отличие от class/object, trait может содержать только объявление, без определения

private методы должны быть определены

Нельзя создавать экземпляры trait, можно наследовать class/object от trait (ключевые слова extends/with)

```
trait traitErr {
//Err: only declaration of private method
  private def privateFoo
}
trait traitOk {
  private def privateFoo(s: String)
                = println(s"I`m $s!")
  def pubFoo() = privateFoo(str)
  val str = "trait Ok"
}
class derived extends traitOk {
  override val str = "class derived"
}
(new derived).pubFoo
//Out: I`m class derived
```

IV. Интерфейсы

abstract class

abstract class, также как и trait:
- может содержать только объявления методов, а также может содержать определения полей и методов
- private методы должны быть определены
- Нельзя создавать экземпляры, можно наследовать class/object

В отличие от trait, для abstract class может быть определён конструктор

Только от одного abstract class можно наследоваться (только ключевое слово extends)

```
abstract class base(i: Int,
                    val str: String) {
  def this(i: Int) = this(i, "clazz")
  def hi(): Unit
}
class derived(i: Int) extends base(i) {
  override def hi = println(s"$str $i")
  hi
}
new derived(111)
//Out: clazz 111

//Err: ctor in trait
trait traitErr(i: Int)
```

V. Наследование

наследование

Ключевые слова «extends» и «with» определяют наследование сущностей.
При этом, «with» может быть использовано только после «extends» и определяет наследование исключительно от trait

В Scala частично допускается множественное наследование: сущность может быть наследована от одного класса (class/case class/abstract class) и множества trait

```
trait Trait0
trait Trait1
trait Trait2
class Clazz0
class Clazz1
//Err!
class WrongDerived0 extends Clazz0 with Clazz1
//Err!
class WrongDerived1 extends Trait0 with Clazz0
//Ok!
class NiceDerived0  extends Trait0 with Trait1
//Ok!
class NiceDerived1  extends Clazz0 with Trait0
                    with Trait1 with Trait2
```

V. Наследование

override, abstract, super

Методы, перегружаемые в сущности-наследнике помечаются ключевым словом «override»

Ключевое слово «super» предоставляет доступ к полям и методом базовой сущности

Ключевое слово «abstract» может применяться к полям и методам trait.

В таком случае «abstract» обязывает использовать trait в качестве базового интерфейса для сущности, в которой присутствует реализация этого метода

```
abstract class Writer {
  def print(str: String): Unit
}
class ConsoleWriter extends Writer {
  def print(str: String) = println(str)
}
trait Uppercase extends Writer {
  abstract override def print(str: String) =
    super.print(str.toUpperCase())
}
trait WithSpaces extends Writer {
  abstract override def print(str: String) =
    super.print(str.toArray.mkString(" "))
}
val writer = new ConsoleWriter with Uppercase
                                with WithSpaces
writer.print("abc") //Out: A B C
```

V. Наследование

полиморфизм при наследовании

Тип экземпляров сущностей наследников может быть (неявно) преобразован к типу базовой сущности

pattern matching может быть использован для уточнения типа экземпляра

```
trait Base {
  def foo = "Base; "}
trait Derived extends Base {
  override def foo = "Derived; "}
object Impl0 extends Base
object Impl1 extends Derived
object Impl2 extends Derived {
  override def foo = "Impl2; "
}
val impls = Seq(Impl0, Impl1, Impl2)
impls.foreach{ impl ⇒
  impl match {
    case Impl0 ⇒ print("Impl0→" + impl.foo)
    case Impl1 ⇒ print("Impl1→" + impl.foo)
    case _ ⇒ print(impl.foo)
  }
} //Out: Impl0→Base; Impl1→Derived; Impl2;
```

VI. Вложенные сущности

вложенные сущности и анонимные экземпляры

В Scala могут быть определены вложенные сущности: trait/abstract class/class/ case class/object, – внутри друг друга

В Scala допускается создание анонимных экземпляров для trait/abstract class. Синтаксис: « new foo(/*ctor params*/) {/*virtual methods implementations*/} »

```
trait traitLevel0 {
  val s: String = ""
  abstract class baseLevel1(s: String) {
    object objLevel2 {
      def say = println(s"$s")
  } }
  def say =
    (new baseLevel1(s) {} ).objLevel2.say
}
val anonymous = new traitLevel0() {
  override val s = "Anonymous"
}
anonymous.say
//Out: Anonymous
```

VII. Self types

self types

Синтаксис «self: T => »
в начале определения сущности

Аналогично механизму наследования, предоставляет доступ к публичным и защищённым полям self type

Объявление self type обязывает наследников также наследоваться от соответствующего типа

```
trait User { protected def name: String }
trait Tweeter { self: User ⇒
  def tweet(msg: String) =
            println(s"$name: $msg")
}
class WrongUser extends Tweeter {
  def name = "WrongUser"
} //Err: Wrong MUST implement User
trait SomeUser extends Tweeter with User {
  override def name = "SomeUser"
}
class User1 extends Tweeter with SomeUser
(new User1).tweet("Hi")
//Out: SomeUser: Hi
```

VII. Self types

self types

Использование self type допускает циклические зависимости

self type позволяет определять структуру экземпляров сущности без объявления методов этой структуры в самой сущности

```
//that`s OK
trait A { self: B ⇒ }
trait B { self: A ⇒ }
//Err: cyclic reference
trait C extends D
trait D extends C

trait Stream { self: {def close: Unit} ⇒
  def flush() = {/*doSomething*/
    close
}}
trait Close {
  protected def close = println("Done!")}
class Impl extends Stream with Close
(new Impl()).flush //Out: Done!
```

VIII. Sealed, алгебраические типы данных

sealed, алгебраические типы данных

Наследники от sealed должны быть объявлены в том же файле

pattern matching: компилятор явно указывает (генерирует warning) на отсутствие типов в матчинге

Алгебраический тип данных – составной тип, представляющий тип-сумму из типов-произведений. Для извлечений значений используется сопоставление с образцом

```
sealed trait Json
case class  JsonString(s: String) extends Json
case class  JsonInt(i: Int) extends Json
case class  JsonBool(b: Boolean) extends Json
object Json {
  def parse(json: Json): Unit = json match {
    case JsonString(s) ⇒ print(s"String $s; ")
    case JsonInt(i) ⇒ print(s"Int $i; ")
    case JsonBool(b) ⇒ print(s"Bool $b; ")
}}
val jsons = Seq(JsonString("Hi"),
                JsonInt(10),
                JsonBool(true))
jsons.foreach(Json.parse)
//Out: String Hi; Int 10; Bool true;
```

That's all Folks!