Лекция 2

Введение в объектно-ориентированное программирование на языке Scala

Синтаксис объявления класса в Scala:

```
class ИмяКласса {
  ...
}
```

Поля, представляющие внутреннее состояние объекта, объявляются в теле класса с помощью конструкций `val` (неизменяемые) или `var` (изменяемые). Например,

```
1 class Point {
2   var x, y, z = 0
3 }
```

Поля обязательно должны быть инициализированы и по умолчанию имеют уровень доступа `public`

Создание объекта (экземпляра класса) выполняется операцией `new`:

```
new ИмяКласса
```

Обращение к полям объекта осуществляется операцией «точка». Например,

```
val p = new Point
p.x = 10
p.y = 20
p.z = 30
```

В примере переменная `p` – неизменяемая в том смысле, что всегда будет указывать на один и тот же объект. Однако, сам объект – изменяемый

Объявление методов в Scala осуществляется конструкцией `def`:

```
def ИмяМетода(формальные параметры): тип = тело
```

Параметры метода – неизменяемые (будто объявлены с помощью `val`)

Методы, не возвращающие значение, объявляются с типом возвращаемого значения `Unit`

Например,

```
1 class Point {
2   var x, y, z = 0
3   def moveBy(p: Point): Unit = {
4     x += p.x
5     y += p.y
6     z += p.z
7   }
8   def sum(): Int = x + y + z
9 }
```

Методы, как и поля, по умолчанию – `public`. Ключевое слово `this` в теле метода обозначает ссылку на объект, у которого вызван метод

Для вызова метода применяется операция «точка», а фактические параметры записываются в круглых скобках:

```
p.moveBy(q)
p.sum()
```

Кроме того, методы могут вызываться как операции:

```
p moveBy q
p sum
```

Более того, «встроенные» в язык операции в действительности являются методами, т.е. `a + b` эквивалентно `a.+(b)`

Компилятор Scala не умеет выводить типы формальных параметров метода. Однако, очень часто он справляется с выведением типа возвращаемого значения. Например,

```
1 class Point {
2   var x, y, z = 0
3   def moveBy(p: Point) = {
4     x += p.x
5     y += p.y
6     z += p.z
7   }
8   def sum() = x + y + z
9 }
```

Если телом метода является блок, внутри блока можно использовать оператор `return` для завершения метода с возвращением значения. Однако, принято обходиться без `return`, т.к. блок и так возвращает значение последнего оператора

В приведённом примере тело метода `moveBy` возвращает `Unit`, т.к. операция присваивания в выражении `z += p.z` возвращает `Unit`

При объявлении методов, возвращающих `Unit`, можно обойтись без знака «=».

```
def moveBy(p: Point) {
  x += p.x
  y += p.y
  z += p.z
}
```

Методы, объявленные без знака «=», всегда возвращают `Unit` вне зависимости от типа значения последнего оператора блока

В языке Scala каждый класс имеет ровно один первичный конструктор и произвольное количество вспомогательных конструкторов

Формальные параметры первичного конструктора записываются в объявлении класса после его имени:

```
class ИмяКласса(формальные параметры) {
  ...
}
```

Тело класса – фактически тело первичного конструктора:

```
1  class Point(px: Int, py: Int, pz: Int) {
2    var x = px
3    var y = py
4    var z = pz
5    def defaults() {
6      x = px
7      y = py
8      z = pz
9    }
10 }
```

Отметим, что первичный конструктор может содержать не только объявления членов класса, но и исполняемые операторы. Например, класс `Point` можно переписать следующим образом:

```
1 class Point(px: Int, py: Int, pz: Int) {
2   var x, y, z = 0
3   def defaults() {
4     x = px
5     y = py
6     z = pz
7   }

9   defaults()
10  if (x == 0 && y == 0 && z == 0) println("origin")
11 }
```

Вспомогательные конструкторы объявляются как методы, имеющие имя `this`, и вызывающие либо первичный, либо другой вспомогательный конструктор с помощью конструкции

```
this(фактические параметры)
```

Например,

```
1 class Point(px: Int, py: Int, pz: Int) {
2   var x = px
3   var y = py
4   var z = pz

6   def this(px: Int, py: Int) = this(px, py, 0)
7   def this() = this(0, 0)
8 }
```

Отметим, что передача фактических параметров конструктору класса осуществляется при вызове операции `new`:

```
new ИмяКласса(фактические параметры)
```

Чтобы запретить вызов первичного конструктора извне класса, нужно поставить ключевое слово `private` перед его списком формальных параметров (работает, начиная с версии 2.10):

```
1 class Point private(px: Int, py: Int, pz: Int) {
2   var x = px
3   var y = py
4   var z = pz

6   def this(px: Int, py: Int) = this(px, py, 0)
7   def this() = this(0, 0)
8 }

10 val p = new Point(10,20,30) // Ошибка!
```

Образец проектирования Singleton имеет в Scala непосредственную поддержку, т.е. можно объявить не класс, а объект с полями и методами:

```
object ИмяОбъекта {
  ...
}
```

Например,

```
1 object Rand1000 {
2   var x = 666
3   def next() = {
4     x = (13*(x + 5)) % 1000
5     x
6   }
7 }
```

Пример использование объекта `Rand1000`:

```
println(Rand1000.x)        // 666
println(Rand1000.next())   // 723
```

Если объект-синглетон и класс имеют одинаковые имена, они становятся компаньонами. Фактически, методы и поля такого объекта-синглетона можно считать статическими методами и полями класса, а его тело – статическим конструктором класса

Класс и объект-синглетон, если они компаньоны, могут обращаться к private-членам друг друга. Например,

```
1 object Point {
2   private var counter = 0
3   def count() = counter
4 }

6 class Point(px: Int, py: Int, pz: Int) {
7   var x = px
8   var y = py
9   var z = pz
10  Point.counter += 1
11 }
```

Операции для класса объявляются как обычные методы. При этом имена методов, соответствующих унарным операциям, начанаются с `unary_`. Например,

```
1 class Point(px: Int, py: Int, pz: Int) {
2   val x = px
3   val y = py
4   val z = pz
5   def unary_- ()    = new Point(-x, -y, -z)
6   def + (q: Point) = new Point(x+q.x, y+q.y, z+q.z)
7   def - (q: Point) = this + (-q)
8   def * (k: Int)    = new Point(k*x, k*y, k*z)
9 }
```

Отметим, что мы сделали объекты класса `Point` неизменяемыми (функциональными), объявив поля класса с помощью конструкции `val`. Хорошим тоном при программировании на Scala считается как можно более широкое использование функциональных объектов

После того, как мы определили операции над объектами класса `Point`, мы можем использовать эти операции в выражениях:

```
val a = new Point(10,15,20)
val b = new Point(20,30,40)
val c = (b - a)*3  // (30,45,60)
```

К сожалению, сейчас мы можем только умножать `Point`'ы на `Int`'ы, но не наоборот:

```
val c = 3*(b - a)  // Ошибка!
```

Для того чтобы это исправить, можно объявить вспомогательный класс `PointFactor` с операцией умножения на `Point` и определить неявное преобразование `Int`'ов в `PointFactor`'ы:

```
class PointFactor(x: Int) {
  def * (p: Point) = p * x
}

implicit def intToFactor(i: Int) = new PointFactor(i)
```