Лекция 3

Механизмы наследования в языке Scala

Предварительное лирическое отступление: в языке Scala можно совместить объявление полей класса с объявлением формальных параметров первичного конструктора

Например, вместо

```
1 class Point2(px: Double, py: Double) {
2   val x = px
3   val y = py
4 }
```

можно записать

```
1 class Point2(val x: Double, val y: Double) {
2 }
```

Более того, пустое тело конструктора можно не записывать, ограничившись объявлением

```
1 class Point2(val x: Double, val y: Double)
```

Как и Java, язык Scala поддерживает одиночное наследование классов:

```
class ИмяКласса(формальныеПараметры)
  extends ИмяБазовогоКласса(фактическиеПараметры) {
  ...
}
```

Тем самым указание базового класса совмещено с вызовом одного из его конструкторов

Пример:

```
1 class Point3(x: Double, y: Double, pz: Double)
2   extends Point2(x, y) {
3   val z = pz
4 }
```

То же самое более лаконично:

```
1 class Point3(x: Double, y: Double, val z: Double)
2   extends Point2(x, y)
```

(Кстати, с точки зрения ОО-проектирования, наследование `Point3` от `Point2` является ошибкой)

Абстрактные классы объявляются с помощью ключевого слова `abstract`. Методы абстрактного класса могут не иметь тел, а поля – значений

Пример:

```
1 abstract class Shape(val color: Int /* Конкретное поле */) {
2   def size: Point2                          // Абстрактный метод
3   def moveBy(delta: Point2): Shape          // Абстрактный метод
4   val square: Double                        // Абстрактное поле
5   def perimeter = (size.x + size.y) * 2     // Кокретный метод
6 }
```

Ещё одно лирическое отступление: методы, не имеющие параметров, можно объявлять, а также вызывать без использования пустых круглых скобок «`()`». Тем самым стирается грань между методами без параметров и `val`-полями

Следует придерживаться соглашения: метод без параметров объявляется и вызывается с использованием «`()`» только в том случае, если он имеет побочные эффекты

Методы и поля базового класса можно переопределять в производном классе. При этом переопределение конкретного метода или поля требует явного указания ключевого слова `override`

Пример:

```
1 class Circle(val p: Point2, val r: Double, color: Int)
2   extends Shape(color) {
3   // Абстрактный метод size определён как поле
4   val size = new Point2(r*2, r*2)

6   // Определение абстрактного метода moveBy
7   def moveBy(delta: Point2) =
8     new Circle(new Point2(p.x + delta.x, p.y + delta.y),
9                r, color)

11  // Определение val-поля square
12  val square = Math.PI * r * r

14  // Переопределение метода perimeter
15  override def perimeter = 2 * Math.PI * r
16 }
```

Чтобы запретить переопределение членов класса, их можно объявить с модификатором `final`

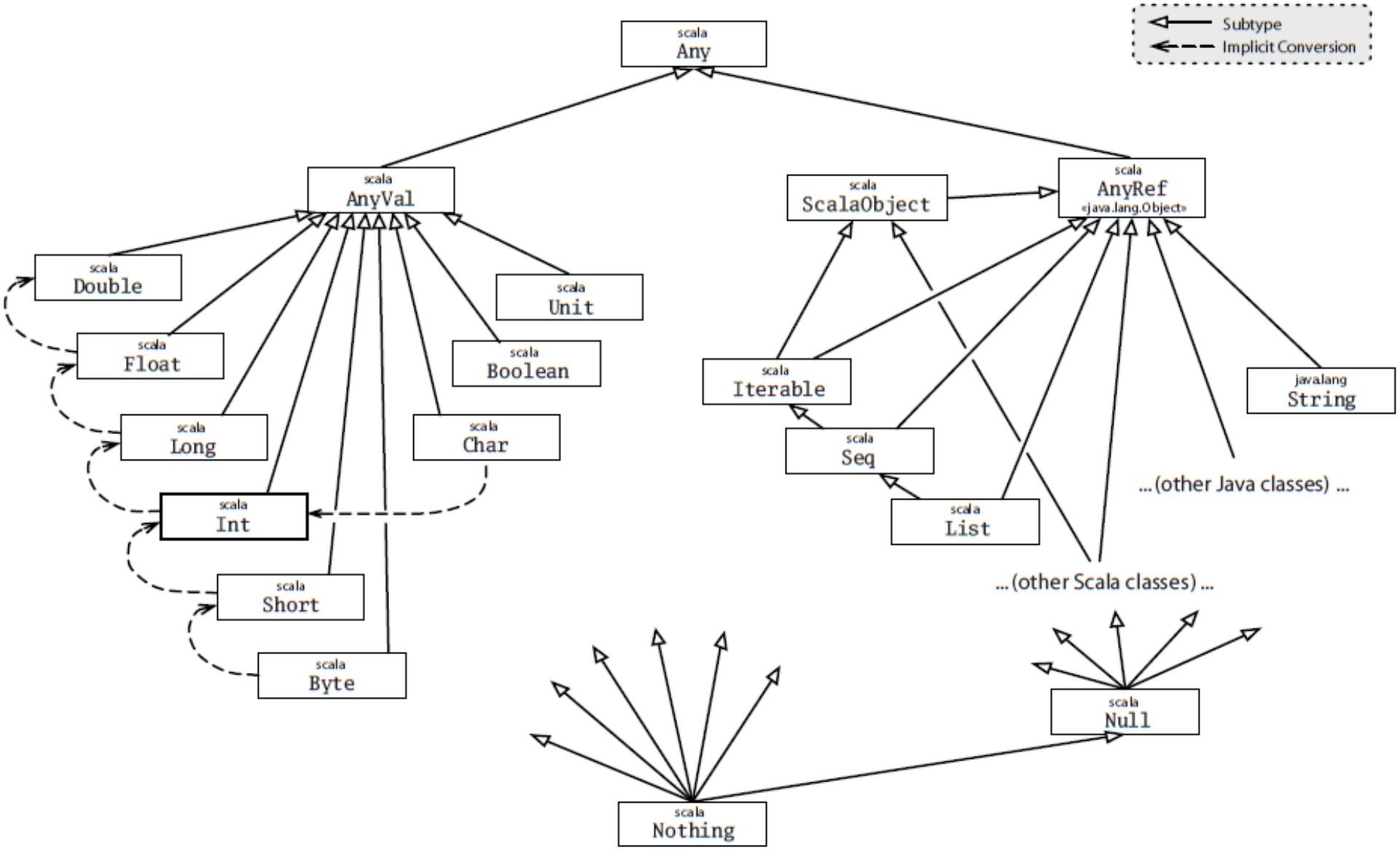
Например, во всех наследниках абстрактного класса `ObjectWithId` равенство объектов будет означать равенство полей `id`, и изменить это никак невозможно:

```
1 abstract class ObjectWithId(val id: Int) {
2   final override def equals(obj: Any) = obj match {
3     case idObj: ObjectWithId => id == idObj.id
4     case _ => false
5   }
6 }

8 class StringWithId(id: Int, val s: String)
9   extends ObjectWithId(id)

11 val a = new StringWithId(666, "abcd")
12 val b = new StringWithId(666, "qwerty")
13 println(a == b)  // true
```

Иерархия классов в Scala



На диаграмме иерархии классов в Scala можно заметить, что, например, класс List имеет сразу два базовых класса: `AnyRef` и `ScalaObject`. Это возможно из-за того, что `ScalaObject` – это на самом деле не класс, а так называемый *трейт* (trait)

Трейт похож на интерфейс Java, но при этом может иметь поля и неабстрактные методы, а также может быть наследником класса. Объявление трейта:

```
trait ИмяТрейта extends БазовыйКласс {
  ...
}
```

Конструктор трейта не может иметь параметров. При этом можно объявить переменную типа трейт, но нельзя создать объект типа трейт

Пример:

```
1 abstract class ObjectWithId(val id: Int)

3 trait EqualsById extends ObjectWithId {
4   override def equals(obj: Any) = obj match {
5     case idObj: ObjectWithId => id == idObj.id
6     case _ => false
7   }
8 }
```

Указание базового класса при объявлении трейта означает, что трейт можно «подмешивать» только к классу — наследнику этого базового класса. «Подмешивание» трейта записывается в объявлении класса после ключевого слова `with`:

```
10 class StringWithId(id: Int, val s: String)
11   extends ObjectWithId(id) with EqualsById

13 val a = new StringWithId(666, "abcd")
14 val b = new StringWithId(666, "qwerty")
15 println(a == b)  // true
```

В любой класс можно подмешать сразу несколько трейтов. Например:

```
10 trait HashById extends ObjectWithId {
11   override def hashCode = id
12 }

14 class StringWithId(id: Int, val s: String)
15   extends ObjectWithId(id) with EqualsById with HashById
```

Более того, подмешивание трейтов можно осуществлять в операции `new`:

```
17 class BoolWithId(id: Int, val b: Boolean)
18   extends ObjectWithId(id)

20 val x = new BoolWithId(667, true)
21   with EqualsById with HashById
22 println(a == x)  // false
```

Подмешивание трейтов похоже на множественное наследование, но лишено сопутствующих проблем («ромбы» в иерархии) благодаря *линеаризациям*

Линеаризация для класса или трейта C – это линейная последовательность базовых классов и трейтов, определяющая, какие именно версии переопределённых методов и полей получает класс или трейт C

Пусть C – класс или трейт, объявленный как
`class/trait` C `extends` C_1 `with ... with` C_n.
Тогда его линеаризация $L\,(C)$ определяется как

$$L\,(C) = C,\ L\,(C_n) \vec{+} \ldots \vec{+} L\,(C_1),$$

где операция $\vec{+}$ – это конкатенация, в которой элементы правого операнда исключают идентичные элементы левого операнда:

$$\begin{aligned} \{a, A\} \vec{+} B &= a, \left(A \vec{+} B\right), && \text{если } a \notin B \\ &= A \vec{+} B, && \text{если } a \in B \end{aligned}$$

Пример:

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

Линеаризация класса `Iter`:

```
Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any
```

Обращение к полю или методу объекта даст ту версию поля или метода, которая получается при поиске слева направо в линеаризации класса этого объекта

Пример:

```
1 class X { def m = "X" }
2 trait A extends X { override def m = "A" }
3 trait B extends X { override def m = "B" }

5 trait P extends X with A with B
6 trait Q extends X with B with A

8 class Test extends X with P with Q
```

Тип	Линеаризация
A	$\mathtt{A}, \mathtt{X}$
B	$\mathtt{B}, \mathtt{X}$
P	$\mathtt{P}, (\mathtt{B}, \mathtt{X}) \vec{+} (\mathtt{A}, \mathtt{X}) \vec{+} \mathtt{X} = \mathtt{P}, \mathtt{B}, \mathtt{A}, \mathtt{X}$
Q	$\mathtt{Q}, (\mathtt{A}, \mathtt{X}) \vec{+} (\mathtt{B}, \mathtt{X}) \vec{+} \mathtt{X} = \mathtt{Q}, \mathtt{A}, \mathtt{B}, \mathtt{X}$
Test	$\mathtt{Test}, (\mathtt{Q}, \mathtt{A}, \mathtt{B}, \mathtt{X}) \vec{+} (\mathtt{P}, \mathtt{B}, \mathtt{A}, \mathtt{X}) \vec{+} \mathtt{X} = \mathtt{Test}, \mathtt{Q}, \mathtt{P}, \mathtt{B}, \mathtt{A}, \mathtt{X}$

```
8 val test = new Test
9 println(test.m) // Печать B
```

Пусть линеаризация класса C выглядит как

$C, X_1, \ldots, X_i, X_{i+1}, \ldots, Any$

Если в классе X_i применена конструкция `super.m`, где `m` – имя метода или поля, то эта конструкция будет обозначать обращение к той версии метода или поля, которую даёт поиск слева направо в линеаризации, начиная с X_{i+1}

Применение конструкции `super.m` даёт возможность реализовывать на языке Scala образец проектирования Decorator путём оформления декораторов в виде трейтов, в определённой последовательности подмешенных к некоторому классу