

Гуляев Г.М.

**Современные технологии
программирования (часть 2)**

Лекция 4. ООП на языке Scala

Курс лекций для студентов АлтГТУ



- ❖ В отличие от java декларация **public** не используется. Все классы в Scala являются публичными.

```
class Counter {  
  private var value = 0  
  def increment { value += 1 }  
  def current = value  
}
```

- ❖ Пример использования класса:

```
val counter = new Counter  
counter.increment  
println(counter.current)
```

- ❖ Для методов без аргументов круглые скобки можно опускать, то есть **Counter()** или **Counter**, **increment()** или **increment**, **current()** или **current** - одно и то же.
- ❖ В некоторых источниках рекомендуют писать скобки, если метод изменяет объект, например, **increment()**, но задумываться об этом - лишняя трата времени. Проще не писать скобки пока компилятор сам их не потребует.

Private переменные, геттеры и сеттеры

❖ Пример класса на java:

```
public class Person {  
    private int age;  
    public int getAge() { return age; }  
    public void setAge(int age) {if (age>this.age) this.age = age; }  
}
```

❖ Реализация того же класса на Scala:

```
class Person {  
    private var privAge = 0  
    def age = privAge  
    def age_= (age: Int) { if (age > privAge) privAge = age }  
}
```

❖ Пример использования:

```
val p = new Person  
p.age = 30  
p.age = 21 // age останется = 30
```

(сеттер в Scala имеет специальный синтаксис: `name_ = (x: A)`)

❖ Если переменная неизменяемая (val), то нет смысла объявлять ее private и писать геттер для нее.

Конструкторы

- ❖ **Вспомогательные конструкторы задаются внутри кода класса при помощи `def this(...)`:**

```
class Person {  
    private var name = ""  
    private var age = 0  
    def this(name: String) { // первый вспомогательный конструктор  
        this // вызываем главный конструктор  
        this.name = name  
    }  
    def this(name: String, age: Int) { //второй вспомог-ый конструктор  
        this(name) // вызываем первый вспомогательный конструктор  
        this.age = age  
    }  
}
```

- ❖ **Примеры использования:**

```
val p1 = new Person // первичный(главный) конструктор  
val p2 = new Person("Иван") // 1-й вспомогательный конструктор  
val p3 = new Person("Иван", 42) // 2-й вспомогательный конструктор
```

Конструкторы

- ❖ Первичный (главный) конструктор задается через указание аргументов в определении класса:

```
class Person (val name: String, val age: Int) {  
    ...  
}
```

- ❖ Если аргументы отсутствуют (как в прошлом примере), то первичный конструктор все-равно есть и он содержит инициализацию переменных в теле класса.

- ❖ При задании переменных можно использовать **val** или **var** или ничего не использовать, также допускается модификатор доступа **private**.

- ❖ Примеры:

```
class Person (name: String, age: Int) {  
    override def toString = name+", возраст: "+age  
}
```

```
val p = new Person("Иван", 42) // Person = Иван, возраст: 42  
p.name // ошибка - нет геттера
```

Конструкторы

❖ Примеры (продолжение):

```
class Person (val name: String, val age: Int)
val p = new Person("Иван", 42) // Person = Person@3f651345
p.name // Иван
p.name = "Федор" // ошибка - нет сеттера
```

```
class Person (var name: String, val age: Int)
val p = new Person("Иван", 42)
p.name // Иван
p.name = "Федор" // String = Федор
p.name // Федор
```

❖ При использовании **val** или **var** создается автоматически закрытая переменная и геттер, а также, в случае **var**, сеттер на нее.

❖ Использование **private** закрывает доступ извне класса к геттеру и сеттеру:

```
class Person (private var name: String, val age: Int) // name недоступна
class Person private (var name: String) // конструктор недоступен
```

Объекты

- ❖ В Scala нет статических полей или методов у класса. Вместо этого используется конструкция **object**.

```
object Index {  
  private var id = 0  
  def getId = { id += 1; id}  
}
```

// при обращении к **Index.getId** будет возвращаться следующее число

- ❖ Конструктор объекта **Index** будет вызван при первом его использовании (когда первый раз вызовем **getId**).

- ❖ Каждая Scala программа стартует в методе **main** какого-либо объекта:

```
object Hello {  
  def main(args: Array[String]) {  
    println("Привет, " + (if (args.length>0) args(0) else "мир")+ "!")  
  }  
}
```

- ❖ Сохраняем в **hello.scala** и выполняем: **scala hello.scala Иван**

Объекты

- ❖ При вызове своей программы через `scala ...` (без предварительной компиляции) не требуется создавать объект с методом `main`:

```
println("Привет, Иван!") // содержание файла hello.scala
```

```
$ scala hello // Привет, Иван (объект был создан автоматически)
```

- ❖ Скомпилировать же этот файл командой `scalac hello.scala` не удастся - для этого нужен объект с методом `main`.
- ❖ После удачной компиляции создаются обычные файлы с расширением `.class` для `jvm`.
- ❖ Для запуска потребуется указать в `classpath` библиотеку `scala-library.jar`:

```
$ java -cp ./srv/server/scala-2.10.0/lib/scala-library.jar Hello Иван
```
- ❖ Можно написать простой `shell` - скрипт `run` (`./run Hello Иван`):

```
#!/bin/sh
java -cp ./srv/server/scala-2.10.0/lib/scala-library.jar $*
exit 0
```


Пакеты и импорт классов

- ❖ **Пакеты в Scala - это просто блоки кода:**

```
package ru {  
  package altailand {  
    package lab10 {  
      object Lab10  
      ...  
    }  
  }  
}
```

- ❖ **Пакеты никак не связаны с файловой системой. Правила видимости как для обычных скобок (внутри пакета видим переменные внешних пакетов).**

- ❖ **Объявление и импорт:**

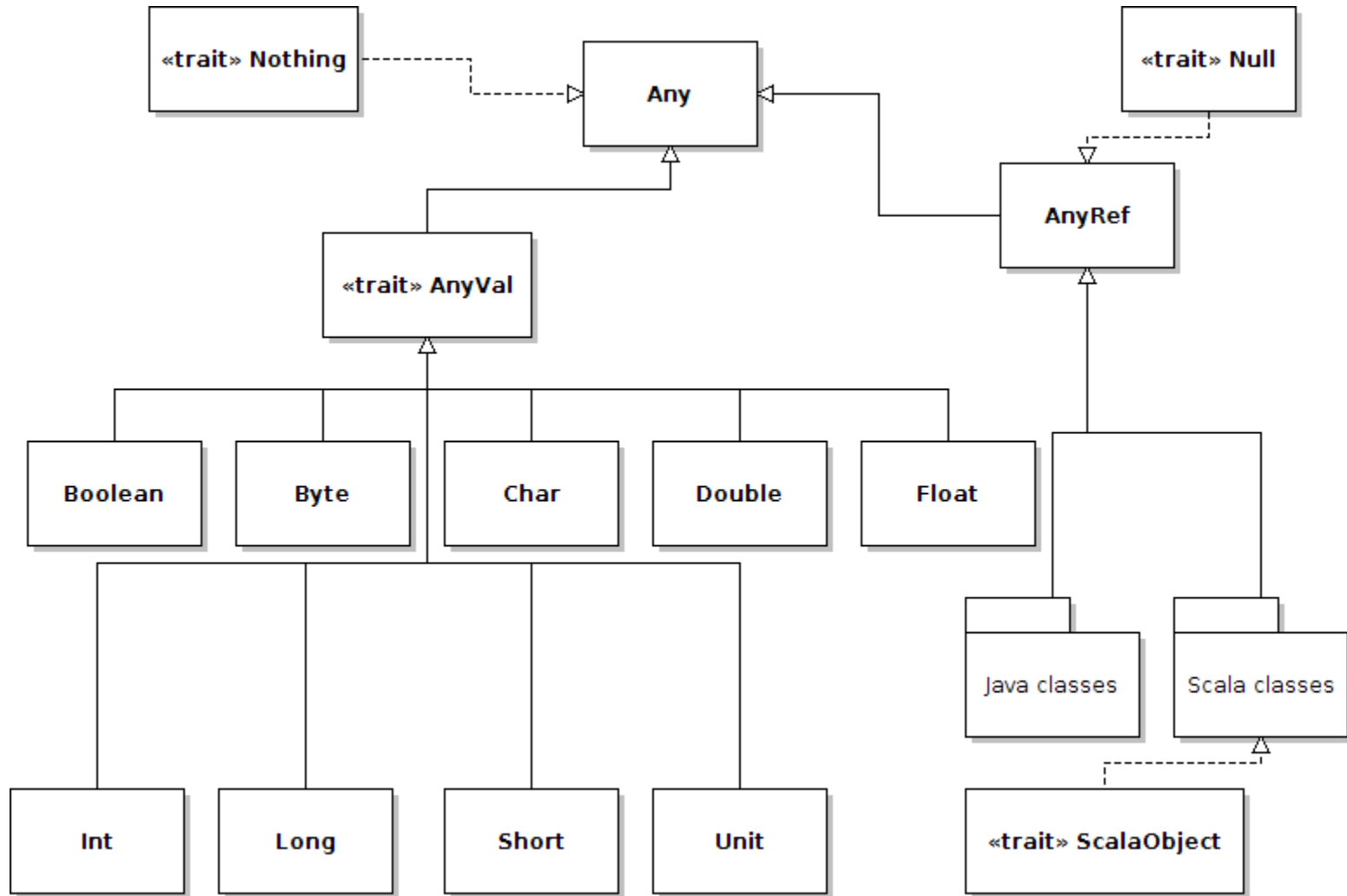
```
package ru.aitailand.lab10 // в начале файла  
import scala.io.Source // в любом месте программы  
import scala.collection.mutable._ // все классы пакета  
import java.awt.{Color, Font} // несколько классов пакета  
import java.util.{HashMap => JavaMap} // переименование класса  
java.lang._ , scala._ , Predef._ // автоматически импортируются
```

Наследование

- ❖ Как и в java одиночное наследование (**extends**):
`class Student extends Person { ... }` // наследование класс ← класс
`object Student extends Person { ... }` // наследование объект ← класс
- ❖ Обязательная директива **override** при перекрытии неабстрактных полей и методов:
`public class Student extends Person {`
 ...
 `override def toString = super.toString + "[group=" + group + "]"`
 // super ссылается на предка Person
}

`class SecretAgent(codename: String) extends Person(codename) {`
 `override val name = "secret"` // прячем настоящее имя
 `override val toString = "secret"` // перекрываем toString
}
- ❖ Использование конструкторов родителя:
`class Student(name: String, group: String) extends Person(name: String)`
`import java.awt.Rectangle`
`class Square(x: Int, y: Int, width: Int) extends Rectangle(x, y, width, width)`

Иерархия наследования



Абстрактные классы

- ❖ Как и в java можно при задании класса использовать ключевое слово **abstract**:

```
abstract class Person(val name: String) {  
    def id: Int // абстрактный метод  
}
```

- ❖ Любой нереализованный метод является абстрактным. Если в классе есть хотя бы один нереализованный метод, то класс является абстрактным.

- ❖ При наследовании к неабстрактному классу все абстрактные методы должны быть реализованы.

```
class Student(name: String) extends Person(name) {  
    def id = name.hashCode // реализация абстрактного метода  
}
```

- ❖ В абстрактном классе могут быть абстрактные поля (их можно переопределять прямо в **new** без наследования):

```
abstract class Person {  
    val id: Int; var name: String // абстрактные поля  
}
```

Трейты

- ❖ Множественное наследование в Scala реализуется при помощи трейтов (ключевое слово **trait**).
- ❖ Трейт занимает промежуточное положение между интерфейсом java и абстрактным классом:
 1. может содержать реализованные методы (как абстрактный класс)
 2. можно наследовать от многих трейтов (как от интерфейсов)
- ❖ Для наследования первого трейта (или класса) используется **extends**, а для остальных слово **with**.
`class Student extends Person with User with Worker ...`

- ❖ **Пример:**

```
trait OddEven {  
  def isOdd: Boolean // абстрактный метод  
  def isEven = !isOdd // не абстрактный метод  
}  
class Number(i:Int) { def isOdd = i%2!=0} // класс без OddEven  
class NumberOE(i: Int) extends Number(i) with OddEven  
val a = new Number(10) with OddEven // добавление трейта к объекту  
val b = new NumberOE(10)  
println(a.isEven+" "+b.isEven) // true true
```

Спасибо за внимание!

www.altailand.ru

