Гуляев Г.М.

Современные технологии программирования

Лекция 5. ООП на языке Scala

Центр компетенции СПО Алтайского края

Классы

- 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 **curent** - 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 a = 0
  def age = a
  def age_= (age: Int) { if (age > a) a = 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 next = { id += 1; id}
}
```

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

- Конструктор объекта **Index** будет вызван при первом его использовании (когда первый раз вызовем **next**).
- 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.altailand.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:

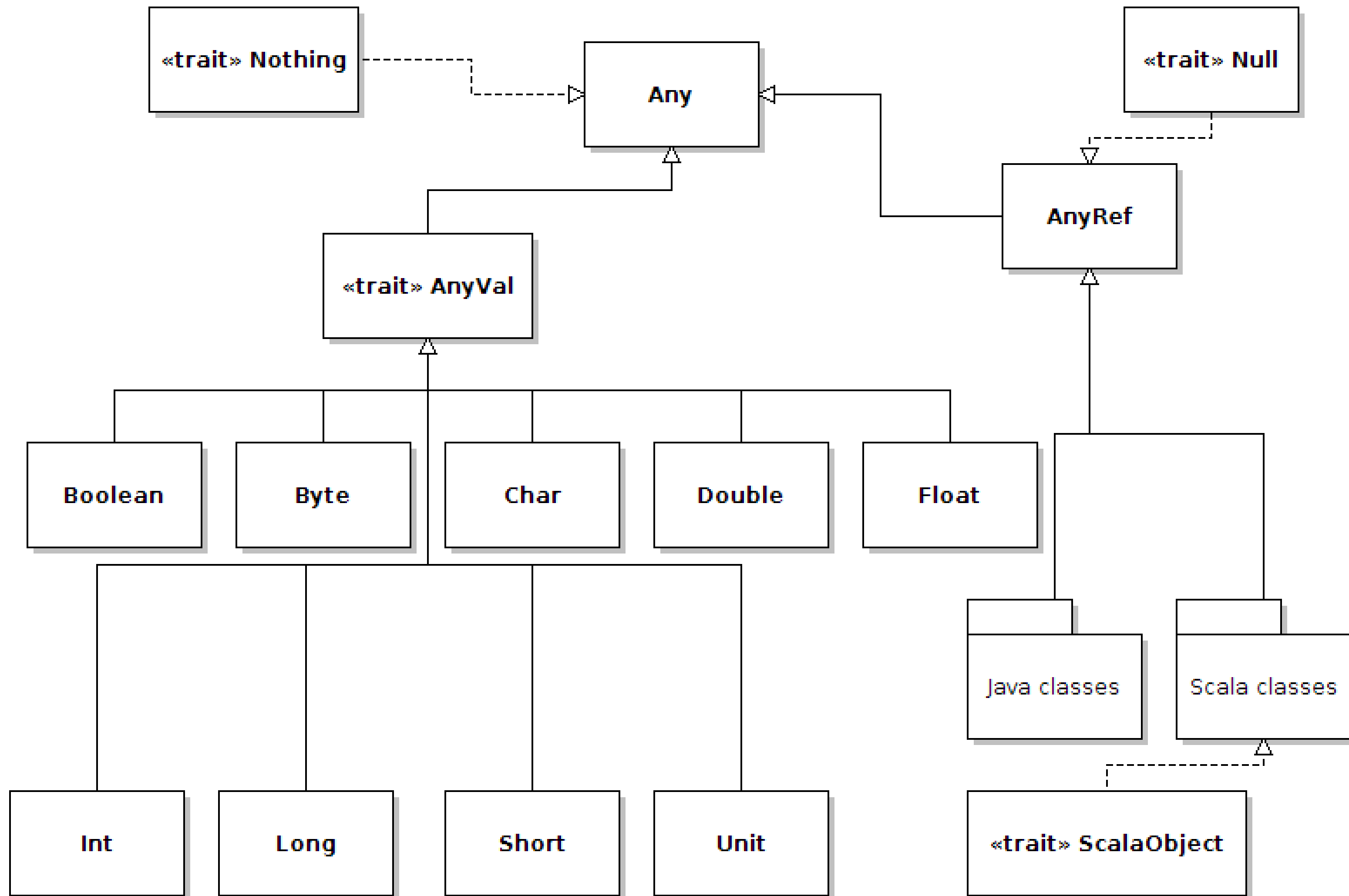
```
public class Student extends Person {
  ...
  override def toString = super.toString + "[group=" + group + "]"
  // super ссылается на предка Person
}

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
```

Равенство объектов

- **Проверка на равенство объектов при помощи ==:**

```
val a =  "abc"
val b =  "abc"
a == b // true (вызывается метод equals затем eq)
```

- **Проверка происходит не по ссылке (как в java), а по значению (внутреннему содержанию объекта):**

```
List(1,2,3,4,5) == (1 to 5).toList // true
```

- **К сожалению, есть исключение для массивов:**

```
Array(1,2,3,4,5) == Array(1,2,3,4,5) // false
```

- **Сравнение основано на методах equals и hashCode. Массивы Scala приводятся к массивам Java и hashCode у них вычисляется по ссылке как в Java.**
- **Для своих классов, без перекрытия этих методов сравнение не работает по значению:**

```
class Person(name: String, age: Int)
val p1 = new Person("Иван",21) // Person@393d3e1d
val p2 = new Person("Иван",21) // Person@45cfaf37
p1 == p2 // false
```

Равенство объектов

- Переопределим методы equals и hashCode:

```
class Person(name: String, age: Int) {
  override def hashCode = 13*name.hashCode+age.hashCode*17
  override def equals(other: Any) = {
    val that = other.asInstanceOf[Person]
    if (that == null) false
    else hashCode == that.hashCode // name==that.name&&age==that.age
  }
}
```

- Теперь уже сравнение происходит так как нужно:

```
val p1 = new Person("Иван",21) // Person@1904abe0
val p2 = new Person("Иван",21) // Person@1904abe0
p1 == p2 // true
```

- Проверка на равенство уже будет работать если переопределить только equals. Однако hashCode используется во многих других случаях, например, при работе со списками из данных объектов. Поэтому, если уж переопределять, то оба этих метода.

Case - классы

- В Scala существует возможность сгенерировать методы equals и hashCode (а также toString) автоматически:

```
case class Person(name: String, age: Int)
val p1 = Person("Иван",21) // Person(Иван,21)
val p2 = Person("Иван",21) // Person(Иван,21)
p1 == p2 // true
p1.hashCode // 778101263
p2.hashCode // 778101263
p1.name // Иван
p1.copy(name="Федор") // Person(Федор,21)
```

- Это так называемые case-классы. Отметим еще необязательность new при создании объектов таких классов и автоматически создаваемые геттеры переменных конструктора без объявления val.
- Case - классы убирают однообразные рутинные операции и уменьшают размер кода, делая его более понятным.

Case - классы

- Пример (комплексные числа):

```
case class Complex(x: Double, y: Double) {
  def mod = math.sqrt(x*x+y*y) // модуль
  def unary_~ = Complex(x,-y) // сопряженное число
  def norm = Complex(x/this.mod,y/this.mod) // нормализация
  def +(z: Complex) = Complex(x+z.x, y+z.y) // сложение
  def -(z: Complex) = Complex(x-z.x, y-z.y) // вычитание
  def *(z: Complex) = Complex(x*z.x-y*z.y, x*z.y+y*z.x) // умножение
  def /(z: Complex) = this * ~z.norm // деление
  override def toString = x + (if (y==0) "" else  " + "+y+"i")
}
```

- Использование:

```
val i = Complex(0,1) // 0.0 + 1.0i
i * i // -1.0
val z1 = Complex(1,1) // 1.0 + 1.0i
val z2 = ~z1 // 1.0 - 1.0i
z1 + z2 // 2.0
z1 * z2 // 2.0
z1 / z2 // 0.0 + 1.414213562373095i
```

Задания для самостоятельной работы

- **Спроектировать и реализовать систему классов для реализации векторной алгебры в трехмерном пространстве (равенство, сложение, вычитание, умножение на число, модуль, скалярное произведение, угол между векторами).**
- **Обобщить предыдущую задачу на n-мерное пространство.**

Спасибо за внимание !