

## Лабораторная работа №3

### Исследование способов реализации классов и объектов в языке Scala

#### 1. Цель работы

Исследовать особенности реализации классов, объектов и трейтов в языке Scala. Реализовать программные модули для обработки сложных структур данных, используя объектно-ориентированный и функциональный подходы.

#### 2. Основные положения

##### 2.1. Классы

##### 2.1.1. Базовое определение и использование классов

В общем виде классы на языке Scala выглядят очень похоже на классы на языке Java:

```
class Counter {  
  private var value: Int = 0  // поля должны инициализироваться  
  
  def increment() {  
    value += 1  
  }  
  
  def current() = value  
}
```

Создание и использование класса полностью аналогично привычным языкам программирования:

```
val myCounter = new Counter  // или new Counter()  
myCounter.increment()  
println(myCounter.current)  // или println(myCounter.current())
```

В языке Scala существует соглашение, согласно которому методы-мутаторы (методы, которые изменяют что-либо, имеют побочный эффект) вызываются с пустыми скобками (в примере это метод `increment()`), а методы-акцессоры (методы, которые возвращают значение и не имеют побочных эффектов) вызываются без скобок (в примере это метод `current`).

Можно насильно заставить использовать синтаксис вызова метода без скобок, если опустить скобки в объявлении метода:

```
...  
def current = value  // объявляем без скобок  
...  
println(myCounter.current())  // вызов со скобками, ошибка компиляции
```

##### 2.1.2. Свойства и методы доступа к ним

В языке Java не принято использовать публичные поля. Взамен этого создаётся приватное поле `value` и пара публичных методов `getValue`, `setValue`:

// это Java:

```
public class Person  
{  
  private int age;
```

```

public int getAge()
{
    return age;
}

public void setAge(int newAge)
{
    this.age = newAge;
}
}

```

Такой подход позволяет при необходимости добавлять логику в работу со свойством. Например, ограничить изменение возраста в меньшую сторону (нельзя омолодить человека):

// это Java:

```

public class Person
{
    private int age;

    public int getAge()
    {
        return age;
    }

    public void setAge(int newAge)
    {
        if (newAge > this.age) { // изменяем возраст только на БОЛЬШИЙ
            this.age = newAge;
        }
    }
}

```

В Scala методы чтения и записи создаются автоматически для каждого поля класса.

Например:

```

class Person {
    var age = 0
}

```

Данный код скомпилируется в класс, содержащий приватное поле age и методы чтения и записи. Эти методы будут публичными (т.к. поле age не было объявлено приватным, для приватного поля будут созданы приватные методы чтения и записи).

В Scala метод чтения получит имя age, а метод записи получит имя age\_=. Пример:

```

val john = new Person()
println(john.age) // вызовет метод john.age()
john.age = 32     // вызовет метод john.age_=(32)

```

Методы чтения и записи можно свободно переопределять на своё усмотрение:

```

class Person {
    private var privateAge = 0

    def age = this.privateAge

    def age_=(newAge: Int): Unit = {
        if (newAge > privateAge) {
            this.privateAge = newAge
        }
    }
}

```

```
}  
}  
}
```

Итого в Scala на выбор существует четыре варианта реализации свойства класса:

- 1) `var attribute` – Scala создаёт методы для чтения и для записи.
- 2) `val attribute` – Scala создаёт только метод чтения.
- 3) Самому определить методы `attribute` и `attribute_`.
- 4) Самому определить метод `attribute`.

### 2.1.3. Конструктор и дополнительные конструкторы

В Scala класс может иметь произвольное количество конструкторов. Однако есть один конструктор, который важнее других – главный конструктор. Кроме этого, класс может иметь любое количество дополнительных конструкторов.

Для начала рассмотрим дополнительные конструкторы. Они имеют два отличия от конструкторов в других языках:

1) Дополнительные конструкторы называются `this` (в Java или C++ конструкторы имеют имя класса)

2) Каждый дополнительный конструктор должен начинаться вызовом одного из дополнительных конструкторов, объявленных выше.

Следующий класс имеет два дополнительных конструктора:

```
class Person {  
  private var name: String = "init"  
  private var age: Int = 0  
  
  def this(name: String) { // дополнительный конструктор  
    this() // вызов главного конструктора  
    this.name = name  
  }  
  
  def this(name: String, age: Int) { // другой дополнительный конструктор  
    this(name) // вызов предыдущего дополнительного конструктора  
  }  
}
```

О главном конструкторе будет рассказано чуть позже. Пока что достаточно знать, что класс, не определяющий главный конструктор явно, получает главный конструктор без аргументов.

Создать экземпляр вышеописанного объекта можно тремя способами:

```
/*  
 * главный конструктор  
 * name = "init"  
 * age = 0  
 */  
val p1 = new Person  
  
/*  
 * первый дополнительный конструктор  
 * name = "Hamilton"  
 * age = 0  
 */  
val p2 = new Person("Hamilton")
```

```
/*
 * второй дополнительный конструктор
 * name = "Jefferson"
 * age = 42
 */
```

```
val p3 = new Person("Jefferson", 42)
```

Главный конструктор на определяется, как метод `this`, а является частью определения класса.

Параметры главного конструктора перечисляются сразу после имени класса:

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

Параметры главного конструктора автоматически превращаются в поля класса, которые инициализируются аргументами конструктора. Выше написанное определение класса `Person` равносильно следующему коду на Java:

// это Java:

```
public class Person
{
    private String name;
    private int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String name()
    {
        return this.name;
    }

    public int age()
    {
        return this.age;
    }
}
```

Главный конструктор выполняет все инструкции в определении класса. Например:

```
class Person(val name: String, val age: Int) {
    println("Создан экземпляр класса Person")
    def description = name + " is " + age + " years old"
}
```

Инструкция `println` является частью главного конструктора и будет вызвана при каждом создании экземпляра класса `Person`. Это удобно, если нужно настроить поле объекта в процессе создания:

```
class MyClass {
    private val someResource = new Resource
    someResource.load(...)
    // ...
}
```

Если после имени класса отсутствуют параметры, то класс получит главный конструктор без параметров, который при создании просто выполнит все инструкции внутри класса.

Так же можно задавать значение параметров класса по-умолчанию (и тем самым избавиться от необходимости задавать дополнительные конструкторы):

```
class Person(name: String = "John Doe", age: Int = 42)
```

## 2.2. Объекты

### 2.2.1. Объекты-одиночки

В Scala отсутствуют статические методы и поля. Вместо этого используются встроенные в язык singleton объекты. Например:

```
object Account {  
  private var count: Int = 0  
  
  def getUniqueID(): Int = {  
    this.count += 1  
    this.count  
  }  
}
```

// пример использования

```
val nextID = Account.getUniqueID()
```

Данный код аналогичен следующему коду на Java:

// это Java:

```
public class Account  
{  
  private static int count = 0;  
  
  public static int getUniqueID()  
  {  
    this.count += 1;  
    return this.count;  
  }  
}
```

Конструктор объекта вызывается при первом обращении к нему в коде. Если к объекту нет обращений в коде – конструктор не вызывается и объект не используется.

Объект обладает всеми свойствами класса, он может наследовать другие классы и трейты (аналоги интерфейсов в Scala). Единственное ограничение – объект не может иметь конструктор с параметрами.

### 2.2.2. Объекты-компаньоны. Общая характеристика

В Java или C++ часто имеется класс, который определяет не только поля методы экземпляра класса, но и статические поля и методы. В Scala добиться такого же эффекта можно с помощью объекта компаньона – объекта, который имеет такое же имя, как и класс.

```
class Account {  
  val id = Account.newUniqueID()  
  private var balance = 0.0  
  
  def deposit(amount: Double) {
```

```

    balance += amount
  }

  ...
}

object Account {
  private var count: Int = 0

  def getUniqueID(): Int = {
    this.count += 1
    this.count
  }
}

```

По сравнению с обычными объектами, объекты-компаньоны имеют два отличия:

1) Их имя совпадает с именем класса.

2) Объект-компаньон имеет доступ к приватным полям и методам класса и наоборот: класс имеет доступ к приватным полям и методам объекта компаньона.

### 2.2.3. Объекты и метод apply

Зачастую в объектах определяется метод `apply`. Он используется для упрощения синтаксиса создания нового экземпляра класса. Например:

// обычное создание экземпляра класса

```

class MyClass {
  var count: Int = 0

  def this(initCount: Int) = {
    this()
    this.count = initCount
  }
}

```

`val foo = new MyClass(13)` // создаст экземпляр класса, где `count = 13`

// создание экземпляра класса с помощью метода `apply` объекта

```

object MyClass {
  def apply(initCount: Int): MyClass = {
    new MyClass(initCount)
  }
}

```

`val bar = MyClass(42)` // отличие в отсутствии `new` перед именем класса

Что мы сделали во фрагменте кода выше? По сути мы сделали обёртку для обычного создания экземпляра класса (с помощью ключевого слова `new`). Но из-за особенности работы языка Scala с методом `apply` (см. лабораторную работу №1) нам удалось избавиться от необходимости использовать ключевое слово `new` каждый раз при создании нового экземпляра объекта. Это может облегчить читаемость кода в сложных выражениях. Например, сравните два варианта создания вложенных массивов:

// Классический вариант с использованием ключевого слова `new`:

```

val matrix = new Array(new Array(1, 2, 3), new Array(4, 5, 6))

```

```
// Вариант с использованием метода apply объекта-компаньона
val matrix2 = Array(Array(1, 2, 3), Array(4, 5, 6))
```

Стоит отметить, что для использования данной возможности метода `apply` объекта, имя объекта не обязательно должно совпадать с именем класса. Пример:

```
object DifferentName {
  def apply(initCount: Int): MyClass = {
    new MyClass(initCount)
  }
}
```

```
val foobar = DifferentName(7) // вернёт экземпляр класса MyClass
```

### 2.3. Трейты

Трейт в языке Scala можно использовать точно так же, как и интерфейсы в языке Java:

```
trait Logger {
  def log(msg: String) // абстрактный метод
}
```

В данном случае не требуется явно указывать, что метод является абстрактным. Нереализованные в трейтах методы автоматически являются абстрактными.

Реализация интерфейса:

```
class ConsoleLogger extends Logger { // extends, не implements
  def log(msg: String) {
    println(msg)
  }
}
```

При переопределении абстрактных методов трейта не требуется указывать ключевое слово `override`.

Если требуется унаследовать более одного трейта, дополнительные трейты добавляются через ключевое слово `with`:

```
class MyClass extends TraitOne with TraitTwo with TraitThree
```

Как и в Java, в Scala класс может быть унаследован от одного класса, но от многих трейтов.

Помимо описанного выше, трейты обладают более обширными возможностями, по сравнению с обычными интерфейсами (например, трейты могут иметь реализацию методов по-умолчанию). Эти особенности, при желании, можно изучить самостоятельно.

## 3. Задание на лабораторную работу

3.1. Реализовать структуру бинарного дерева поиска (элементы в левом поддереве меньше, чем элемент в корне, а элементы правого поддерева больше, чем элемент в корне). Для реализации:

3.1.1. Написать трейт `Tree`, определяющий абстрактные методы `getLeftSubtree: Tree`, `getRightSubtree: Tree` и `getNodeData: Int`.

3.1.2. Написать класс `Node`, который реализует трейт `Tree` и представляет собой узел дерева.

3.1.3. Написать класс `Leaf`, который реализует трейт `Tree` и представляет собой лист дерева.

3.2. Для реализованной структуры дерева написать следующие функции:

- 3.2.1. Функцию `printTree(Tree): Unit`, которая выводит на экран дерево
- 3.2.2. Функцию `insert(Int, Tree): Tree`, которая принимает элемент для вставки и корень дерева, возвращает корень нового дерева со вставленным элементом (при этом изначальное дерево изменяться не должно).
- 3.2.3. Функцию `contains(Int, Tree): Boolean`, которая возвращает `true` или `false` в зависимости от того, содержится ли заданное число в дереве, или нет.
- 3.2.4. Функцию `sum(Tree): Int`, которая возвращает сумму всех элементов в дереве.