Лекция 1

# Введение в функциональное программирование на языке Scala

Особенности языка Scala:

1. Подержка как объектно-ориентированной, так и функциональной парадигм программирования

2. Строгая статическая типизация

3. Компиляция в байт-код Java

4. Наличие среды REPL (Read-Evaluate-Print Loop)

Функциональная парадигма – программа представляет собой набор чистых функций

Чистая функция – это функция, не имеющая побочных эффектов (не обращается к глобальным переменным, не осуществляет ввод/вывод)

Из определения чистой функции следует:

1. она всегда возвращает значение (в противном случае, её вызов не имеет смысла)

2. сколько бы мы не вызывали чистую функцию, передавая ей один и тот же набор фактических параметров, она всегда будет возвращать одно и то же значение (ей больше неоткуда брать данные, кроме как из параметров, поэтому возвращаемое значение полностью определяется значениями параметров)

Запись типов данных в подмножестве языка Scala, которое будет использоваться в данной лекции:

Int – целое число

Boolean – булевское значение

String – неизменяемая строка

List[$T$] – список, элементы которого имеют тип $T$

$(T_1, T_2, \ldots, T_n)$ – кортеж размера $n$, элементы которого имеют типы $T_1$, $T_2$, ..., $T_n$

$(T_1, T_2, \ldots, T_n) => S$ – функция, принимающая $n$ параметров, которые имеют типы $T_1$, $T_2$, ..., $T_n$, и возвращающая значение типа $S$

Примеры записи типов данных:

1. Список функций, принимающих два целочисленных параметра, и возвращающих строки:

   ```
   List[(Int,Int) => String]
   ```

2. Функция, принимающая целое число, и возвращающая пару целых чисел (т.к. параметр – единственный, скобки вокруг Int можно не писать):

   ```
   Int => (Int,Int)
   ```

3. Функция, принимающая пару из целого числа и строки и возвращающая список целых чисел (двойные круглые скобки необходимы, т.к. без них получится функция, принимающая два целочисленных параметра):

   ```
   ((Int,String)) => List[Int]
   ```

1. Целочисленные константы:

```
0 100 0x40
```

2. Строковые константы:

```
"Hello, World!\n"
```

3. Списковые литералы:

```
Nil                    // Пустой список
List[Int](1, 2, 3) // Список из чисел 1, 2, 3
List(1, 2, 3)          // (тип элемента может быть выведен)
```

4. Кортежные литералы:

```
(10, "alpha", 20)  // Кортеж из трёх элементов
```

5. Функциональные литералы:

```
// Функция, возвращающая сумму двух чисел
(a: Int, b: Int) => a + b
```

Некоторые операции над списками:

```
// Конкатенация
List(1, 2, 3) ::: List(4, 5, 6)

// Добавление элемента в начало
10 :: List(20, 30, 40)

// Вычисление длины
(1 :: 2 :: 3 :: Nil).length
```

Операция доступа к компонентам кортежа:

```
(10, 20, 30, 40)._2  // возвращает 20
```

Операторы в Scala возвращают значения

Например, оператор if по сути является аналогом тернарной операции языка C:

```
if (x == 0) 1 else x*10
```

Оператор-блок представляет собой последовательность операторов, заключённую в фигурные скобки, и возвращает значение последнего оператора:

```
{
  println("Hello, World!")
  20
} // возвращает 20
```

Операторы в блоке, вообще-то, разделяются с помощью «;», но компилятор Scala в большинстве случаев умеет сам добавлять «;» (как компилятор языка Go)

Функциональная парадигма подразумевает использование связывания переменных и значений, а не присваивания

```
val x: Int = 10
val s, t: String = "hello"
val fact: Int => Int =
  (x: Int) => if (x == 0) 1 else x*fact(x-1)
```

Отличие связывания от присваивания – значение не может быть изменено

Допустимо не указывать тип переменной при связывании, если тип можно вывести из значения:

```
val x = 10
val s, t = "hello"
```

Тип переменной «fact» не может быть выведен. Однако, может быть выведен тип переменной «x» в записи функции:

```
val fact: Int => Int =
  x => if (x == 0) 1 else x*fact(x-1)
```

Конструкцию «val» можно использовать для декомпозиции кортежей, списков и других составных значений

```
val pair = (10, "hello")
val (i, s) = pair   // i == 10, s == "hello"

val list = List(1, 2, 3)
val x :: xs = list  // x == 1, xs == List(2, 3)
```

Выражение слева от «=» называется образцом, а конструкция val в действительности выполняет сопоставление выражения, находящегося справа от «=», с образцом. Образцы могут содержать константы:

```
val 1 :: tail = List(1, 2, 3) // tail == List(2,3)
```

Если бы список справа от «=» не начинался с «1», сопоставление привело бы к порождению исключения «scala.MatchError»

Пример (функция суммирования элементов списка):

```
val sum: List[Int] => Int =
  list =>
    if (list == Nil)
      0
    else {
      val x :: xs = list // декомпозиция списка
      x + sum(xs)
    }
```

Пример (функция замены вхождений «a» на «b»):

```
val replace: (List[Int], Int, Int) => List[Int] =
  (list, a, b) =>
    if (list == Nil)
      Nil
    else {
      val x :: xs = list // декомпозиция списка
      (if (x == a) b else x) :: replace(xs, a, b)
    }
```

Существует специальная форма блока, определяющая так называемую «частичную функцию»:

```
{
  case ...образец... => ...
  case ...образец... => ...
  ...
  case ...образец... => ...
}
```

Аргумент частичной функции сопоставляется по очереди с образцами в конструкциях «case» до первого удачного сопоставления, и тогда функция возвращает значение выражения, расположенного справа от «=»>

Если ни одного удачного сопоставления не получилось, частичная функция порождает исключение «scala.MatchError»

Пример (функция суммирования элементов списка):

```
val sum: List[Int] => Int = {
  case Nil      => 0
  case x :: xs => x + sum(xs)
  }
```

Пример (функция замены вхождений «a» на «b»):

```
val replace: (List[Int], Int, Int) => List[Int] = {
  case (Nil, a, b)      => Nil
  case (x :: xs, a, b) =>
    (if (x == a) b else x) :: replace(xs, a, b)
  }
```

К образцам можно добавлять дополнительные условия («гарды»):

```
val replace: (List[Int], Int, Int) => List[Int] = {
  case (Nil, a, b)                    => Nil
  case (x :: xs, a, b) if (x == a) => b :: replace(xs, a, b)
  case (x :: xs, a, b)                => x :: replace(xs, a, b)
  }
```

В языке Scala широко применяются функции высших порядков, принимающие другие функции в качестве аргумента, а также возвращающие значения типа функция

Пример (сумма элементов списка, удовлетворяющих предикату):

```
val sum: (List[Int], Int => Boolean) => Int = {
  case (Nil, p)                    => 0
  case (x :: xs, p) if (p(x)) => x + sum(xs, p)
  case (x :: xs, p)                => sum(xs, p)
  }
```

Пример (вызов функции sum):

```
val list = List(1, 2, 3, -5, 6, -100)
val s = sum(list, (x: Int) => x > 0)  // s == 12
```

Предикат можно записать короче:

```
val s2 = sum(list, x => x > 0)  // Scala выводит тип x
val s3 = sum(list, _ > 0)       // А можно и так :-)
```

Часто используют приём, называемый «закарриванием», при котором функция, имеющая несколько параметров, переписывается так, чтобы принимать часть параметров и возвращать функцию, принимающую остальные параметры

Пример (сумма элементов списка, удовлетворяющих предикату):

```
val sum: (Int => Boolean) => (List[Int] => Int) =
  p => {
    case Nil                  => 0
    case x :: xs if (p(x)) => x + sum(p)(xs)
    case x :: xs              => sum(p)(xs)
  }
```

Пример (вызов функции sum):

```
val list = List(1, 2, 3, -5, 6, -100)
val sumPositive = sum(_ > 0)
val s = sumPositive(list)  // s == 12
```