Программирование на Scala

БАЗОВЫЕ КОНЦЕПЦИИ

I. Условные выражения и циклы

If expression

Тривиальный синтаксис, схожий с Java/C#/C++

```
if (trueOrFalse) {
  doSomething()
} else if (falseOrTrue) {
  doSomethingElse()
} else {
  doAnotherThing()
}
```

Условный переход представляет собой выражение (expression)

```
def abs(i: Int) = if (i < 0) -i else i
```

Сравнение тернарного оператора Java и условного выражения Scala:

```
JAVA: int i = a > b ? a : b

SCALA: val i = if (a > b) a else b
```

I. Условные выражения и циклы

for loop

В Scala существует несколько вариантов синтаксиса объявления цикла for

Так, например, Scala поддерживает синтаксис вида for (element <- iterable)

```
val strings = Seq("A", "B")
val numbers = Seq(1, 2, 3)

for (str ← strings) print(str + "; ")
//Out: A; B;

for (num ← numbers) print(num + "; ")
//Out: 1; 2; 3;

for {str ← strings
     num ← numbers} print(str + num + "; ")
//Out: A1; A2; A3; B1; B2; B3;
```

I. Условные выражения и циклы

for loop

Использование генераторов в циклах for

```
for (i ← 1 to 5) print(s"$i; ")
//Out: 1; 2; 3; 4; 5;

for (i ← 1 until 5) print(s"$i; ")
//Out: 1; 2; 3; 4;
```

Использование “гардов” (guards) в циклах for

```
for { i ← 1 to 5
      if (0 == i % 2) } print(s"$i; ")
//Out: 2; 4;

for { i ← 1 to 5
      if (0 == i % 2)
      if (i > 2) } print(s"$i; ")
//Out: 4;
```

I. Условные выражения и циклы

for loop

Конструкция for-yield;
for как выражение
(expression)

```
val strings = Seq("A", "B")
val res = for {i ← 1 to 2
               str ← strings
               } yield (s"$str$i")
print(res)
//Out: Vector(A1, B1, A2, B2)
```

for эквивалентен foreach:

```
//(1):
for { i ← 1 to 2 } print(i)
//(2):
(1 to 2).foreach(i ⇒ print(i))

//(1) = (2)
```

for-yield эквивалентен map:

```
val res0 = for {i ← 1 to 2}
           yield (i * i)

val res1 = (1 to 2).map{i ⇒ i * i}
print(res0 == res1) //Out: true
print(res0) //Out: Vector(1, 4)
```

I. Условные выражения и циклы

while/do-while

Синтаксис while/do-while:

```
var i = 0
while (i < 5) {
  print(s"$i; ")
  i += 1
}
//Out: 0; 1; 2; 3; 4;
```

```
var j = 0
do {
  print(s"$j; ")
  j -= 1
} while (j > 0)
//Out: 0;
```

while/do-while в Scala – это выражение (expression), которое возвращает тип Unit

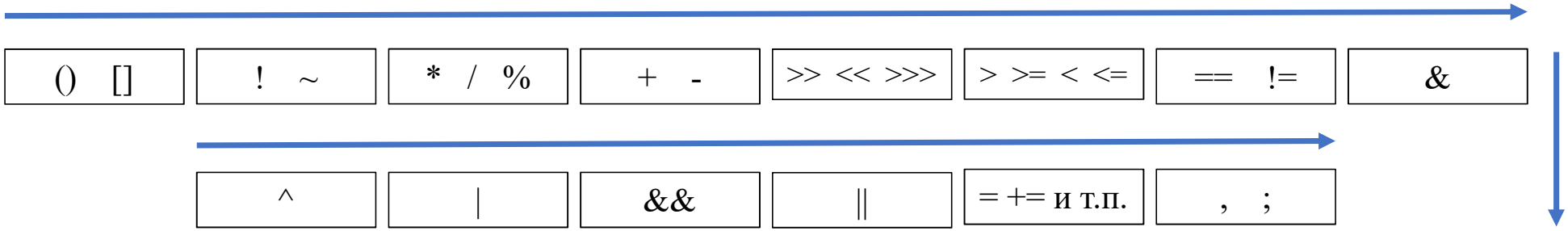
```
var i = 0
val res = while (i < 5) {
  someFunction(i)
  i += 1
}
print(res)
//Out: ()
```

II. Операторы

типы операторов, приоритет операторов

Арифметические операторы:	+ - * / %
Операторы сравнения:	== != > < >= <=
Логические операторы:	&& || !
Битовые операторы:	& | ^ ~ << >> >>>
Операторы присвоения:	= += -= *= /= %= <<= >>= &= ^= |=
Прочие:	() [] , ;

Приоритет операторов (от высшего к низшему):



II. Операторы

оператор=метод, перегрузка операторов

В Scala оператор – это метод

Инфиксное (infix) определение операторов:

```
val sum = 1 + 2.+(3)
print(sum)
//Out: 6
val mul = 3.*(2) * 2
print(mul)
//Out: 12
```

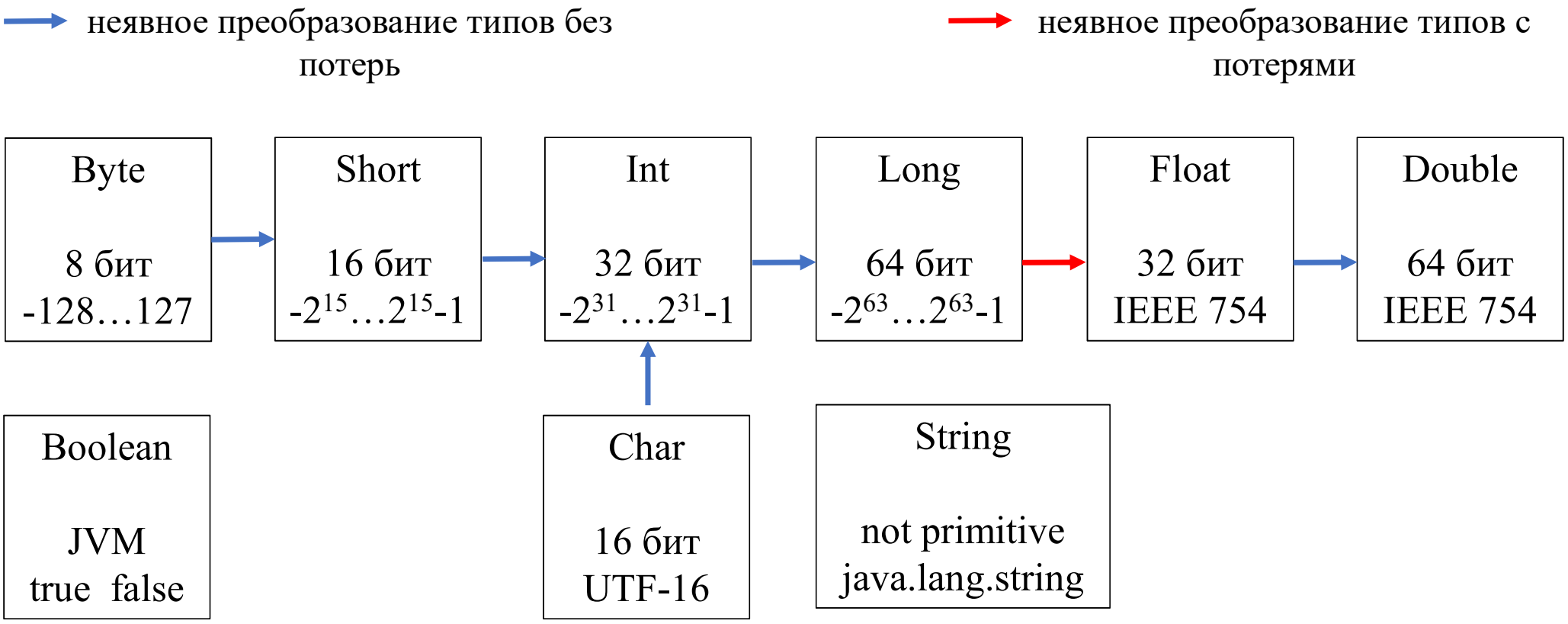
Перегрузка операторов:

```
case class Vector2D(x: Double, y: Double) {
  //scalar multiplie
  def * (vec2D: Vector2D) =
      x * vec2D.x + y * vec2D.y
}
val e = Vector2D(1.0, 1.0)
val mul = Vector2D(-2.0, 3.0) * e
print(mul)
//Out: 1.0
```

III. Базовые типы

перечисление примитивных типов и String, неявное преобразование типов

Примитивные типы в Scala и тип String:



III. Базовые типы

объявление переменных типа, явное преобразование типов, литералы

Присвоение по умолчанию:
Целое число = Int
Дробное число (с символом “.”, например, 3.0) = Double

Явное преобразование:
toByte toShort toInt toLong toChar toFloat toDouble toString ~~toBoolean~~

Литералы:
prefix **0x** = hex integer
postfix **L** = Long
postfix **F** = Float
‘’ = Char “” = String

```
val b0: Byte = 0
val b1: Byte = 0x7F //127
val s0: Short = 127
val s1: Short = 255
print(s0.toByte)
//Out: 127
print(s1.toByte)
//Out: -1, т.к. 255 за границами Byte
val l = 1000L //l: Long
val i = 0x7F //i: Int
val d = 0.0 //d: Double
val f = 0.0F //f: Float
val ch = 'A' //ch: Char
val str = "SomeString" //str: String
```

IV. Функции

объявление функций

Сигнатура функции:

```
def foo(arg0: Type0, arg1: Type0, arg2: Type1): Type1 = {...}
```

Функции могут быть объявлены внутри сущностей class, trait, object, а также внутри других функций

Функции могут содержать параметры по умолчанию

Функции могут быть анонимными

```
object Foo {
  def say(number: Int): Unit = {
                          //defaul value
      def sum(i0: Int, i1: Int = 10)/* :Int */ = {
        i0 + i1
      }
      val sayAnonimous =
        (str: String) ⇒ print("Anonymous-" + str)
      //anonymous function
      sayAnonimous(sum(number).toString)
  }
}
Foo.say(1)
//Out: Anonymous-11
```

IV. Функции

именованные аргументы, каррирование (currying)

Именованные аргументы. Можно поимённо передавать аргументы в функцию, при этом можно нарушать порядок аргументов

```
def fourWords(w0: String = "zero",
              w1: String= "one",
              w2: String = "two",
              w3: String = "three"): String
    = s"$w0 $w1 $w2 $w3"

print(fourWords(w3 = "THIRD", w1 = "FIRST"))
//Out: zero FIRST two THIRD
```

Каррирование (currying). Можно определять более одного списка аргументов функции

```
def sum2(x: Int) = (y: Int) ⇒ x + y
def sum2_currying(x: Int)(y: Int) = x + y
def sum3_currying(x: Int)(y: Int)(z: Int) = x + y + z
print(sum2(1)(2)) //Out: 3
print(sum2_currying(1)(2)) //Out: 3
print(sum3_currying(1)(2)(3)) //Out: 6
```

IV. Функции

функции высшего порядка

Сигнатура функции-аргумента:

```
(inputType0, inputType1) ⇒ outputType
```

Функции могут выступать в качестве параметра других функций

Функции могут возвращать другие функции

Такие функции называются **функциями высшего порядка**

```
def getNumbers(): Seq[Int] = Seq(11, 22)
def oddOrEven(generator: () ⇒ Seq[Int],
              index: Int): (String) ⇒ String = {
  def sayOdd(str: String) = s"$str is odd; "
  def sayEven(str: String) = s"$str is even; "
  val numbers = generator()
  val num = numbers(index)
  if (num % 2 == 0) sayEven else sayOdd
}
print(oddOrEven(getNumbers, 0)("That number"))
print(oddOrEven(getNumbers, 1)("And that"))
//Out: That number is odd; And that is even;
```

IV. Функции

call-by-value, call-by-name

call-by-value передаваемый аргумент известен перед выполнением функции

call-by-name вычисляет передаваемое выражение **каждый раз при разыменовывании** аргумента

call-by-name синтаксис отличается объявлением параметра функции:

def foo(**expr: => Type**)

```
def callByValue(i: Int) = {
  print(s"i ⇒ $i; ")
  print(s"ii ⇒ $i; ") }
def callByName(i: ⇒ Int) = {
  print(s"i ⇒ $i; ")
  print(s"ii ⇒ $i; ") }
var someInt = 2
callByValue {
  someInt += 1
  someInt * someInt
} //Out: i ⇒ 9; ii ⇒ 9;
callByName {
  someInt += 1
  someInt * someInt
} //Out: i ⇒ 16; ii ⇒ 25;
```

V. Коллекции (введение)

кортежи (tuples), списки (lists), maps

В стандартной библиотеке Scala реализованы **все базовые коллекции**:
списки (линейные и индексированные), **множества**, **maps**, а также **кортежи**

Неизменяемые (**immutable**) и изменяемые (**mutable**) коллекции
По умолчанию коллекции в Scala – **неизменяемые (immutable)**, если явно не указан package с изменяемыми коллекциями

Коллекции в Scala имеют общий интерфейс **Iterable**
В Iterable определены методы, обеспечивающие удобство работы с коллекциями (например, map)
Часть методов определена только для конкретных коллекций и отсутствует в Iterable (например, sortBy)

V. Коллекции (введение)

кортежи (tuples)

Кортежи представлены посредством сущностей
Tuple2, Tuple3 … Tuple22

Можно объявлять нужный Tuple{X} посредством синтаксиса
(elem1, elem2, … elem{X})

Доступ к элементам:
_1, _2, _3 … _{X}

pattern matching

```
val tup0 = Tuple2(0, "Zero")
val tup1 = Tuple3(1.0F, 1, "One")
val tup = (2.0F, 2, "Two", tup0, tup1)
//tup = Tuple5( ... )
```

```
val square = ("Square", 2.0F, 4.0F)
val cube = ("Cube", 1.0F, 2.0F, 3.0F)
val area = square._2 * square._3
val volume = cube._2 * cube._3 * cube._4
List(square, cube).map{
  case ("Square", _, _) ⇒ print(area)
  case ("Cube", _, _, _) ⇒ print(volume)
}
```

V. Коллекции (введение)

списки (lists)

В стандартной библиотеке Scala реализованы несколько видов списков, например, **List**

for loop,
так как List – есть реализация Iterable

Операторы конкатенации с элементами:
+: (prepend) и **:+ (append)**;
с другим списком: ++;
при этом (для **immutable**) **создаётся новый контейнер**

```
val listInt = List(1, 2, 3)
val listStr = List("One", "Two", "Three")
/*val list = List(1, "One") ERROR! */

for (el ← listStr) print(s"$el; ")
//Out: One; Two; Three;
for (i ← 1 to 2) print(s"${listInt(i)}; ")
//Out: 2; 3

val listStr1 = listStr ++ List("IV", "V")
val listStr2 = listStr1 :+ "VI"
val listStr3 = "0" +: listStr2
println(listStr3)
//Out: List("0", "One", "Two", "Three", "IV", "V")
```

V. Коллекции (введение)

maps

В стандартной библиотеке Scala **Map** – общий интерфейс (trait)

for loop, так как Map – наследник Iterable

Получение элементов **по ключу** – оператор ()

Оператор объединения maps ++

```
val diameters = Map("Mercury" → 4878,
                    "Mars" → 6779,
                    "Earth" → 12742)
for (d ← diameters) print(s"$d;")
//Out:"Mercury"→4878;"Mars"→6779;"Earth"→12742
val diameterOfMars = diameters("Mars")
print(diameterOfMars) //Out: 6779
for ((str, i) ← diameters) print(s"$i; ")
//Out: 4878; 6779; 12742;
val moreDiameters = Map("Pluto" → 2302) ++ diameters
for ((str, i) ← moreDiameters) print(s"$str; ")
//Out: Pluto; Mercury; Mars; Earth;
```

V. Коллекции (введение)

некоторые методы Iterable

Itearable – базовый интерфейс для всех структур данных стандартной библиотеки Scala

В Itearable определены методы, реализованные для структур данных и облегчающие работу с ними

map, foreach, size

Также определены методы **преобразования** между типами контейнеров: **toList, toMap, toSeq, toSet, toArray,**

И множество других полезных методов (**groupBy, flatMap, maxBy, fold, foldLeft, foldRight, head, tail и т. д.**)

```
val diameters = Map("Mercury" → 4878,
                    "Mars" → 6779,
                    "Earth" → 12742)
val radii = diameters.map{ case (str, d) ⇒
  str → (d / 2)
}
radii.foreach{ r ⇒ print(s"$r;") }
//(Mercury,2439);(Mars,3389);(Earth,6371);
println(diameters.size) //Out: 3
val names = diameters.map(_._1).toSeq
print(names.map(_.toUpperCase))
//Out: List(MERCURY, MARS, EARTH)
```

V. Коллекции (введение)

изменяемые коллекции (mutable)

package
scala.collection.mutable

операторы
изменения коллекции
+= ++=
-=

для maps
изменение значения **по ключу**

для списков
изменение значения **по индексу**

```
val list =
  scala.collection.mutable.ListBuffer.empty[Int]
list += 1
list ++= List(2, 3)
list.foreach(i ⇒ print(s"$i; "))
//Out: 1; 2; 3
```

```
val diameters = scala.collection.mutable.Map(
                    "Mars" → 6779,
                    "Earth" → 12742)
diameters += "Uranus" → 51118
diameters("Mars") = -100
diameters ++= Map("A-C" → diameters("Earth"))
diameters -= "Earth"
print(diameters)
//HashMap(Uranus→51118,Mars->-100,A-C→12742)
```