Introduction in Scala

Alexander Podkhalyuzin

September 21, 2012

Control Structures

Intoduction

This part intersects with first lecture. Let's take a look, how you can control flow in your Scala program.

If statement


```
if (condition) expression1 [else expression2]
```

You can use if statement as ternary operator, however you should consider a way to exatract such value for better readability.

While statement


```
while (condition) expression
do expression while (condition)
```

While statement condition should be changed somehow to avoid infinite loop, so you have to use mutable state for while statement.
There better possibility to use tail recursion:

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

Additionally it returns value itself.

For Statement


```
val array = Array(1, 2, 3)
for (x <- array) println(x + 1)
```

This is monad-like syntactic sugar. There is no indexed for statement sytax. You should use range syntax instead:

```
1 to 3
1 until 3
1 to 10 by 2
(1 to 3).reverse
```

For Statement features

- Yielding collection
- Any number of generators
- Filtering feature
- Binding new variables

All of this is just syntax sugar. If class doesn't contain 'withFilter' method you can't use filtering in for statements.

Shadowing scope is between two generators. So for two consequent generators you can bind the same name.

For statement pattern irrefutability

Imagine the List:

```
val list = List((1, 2), (1, 2, 3), 1, "text")
```

Then you are not able to use the following code:

```
for (a: Int <- list) yield a
```

But following code is ok, it filters out, what doesn't match:

```
for ((a: Int, b: Int) <- list) yield a
```

Try statement


```
try expression
[catch partialFunction]
[finally expression]
```

Scala hasn't checked exceptions. So you don't need to use try/catch everywhere. However it's still a way to control your program execution, and it's still the best way to control program exceptions.

Try statement returns value of least upper bound of try clause and catch clause partial function return type.

Throw statement

The same like in Java:

```
throw expression
```

Note, this statement breaks program flow, so type of this statement is Nothing, which is applicable to any other Scala language type.

Unimplemented code syntax:

```
class NewClass {
  def method1 = 1
  def method2 = ???
}
```

Partial Function

In the catch block expression of type PartialFunction[Throwable, T] is required. Actually it requires any type with methods isDefinedAt and apply.

Now we need to know just that the following expression is PartialFunction:

```
try {
} catch {
  case e: IOException =>
  case f: FileNotFoundException =>
}
```

Finally block

Value returned from finally block is ignored, so it's not included into actual type of complete try statement. That's why you can get weird behavior.

ControlThrowable

Scala have special type of Throwable called ControlThrowable. It doesn't have method fillInStacktrack, which causes most of performance degradation on using exceptions to control program flow.

NonLocalReturnControl is such example of ControlThrowable.

Match statement

Let's take a look at match statement as more flexible Java switch statement:

```
args.headOption match {
  case Some("text1") => "text2"
  case Some("text2") => "text1"
  case _ => "default"
}
```

As usual return type of match statement is least upper bound of all case clauses.

break/continue replacements

Scala has no break and continue operators.

- Any while statement with break/continue, you can replace by recursive function.
- If stack size is significant, and you can't replace by tail-recursive, you can replace it using flags.
- The last possibility is to use ControlThrowable for break.

```
import scala.util.control.Breaks._
breakable {
  while (true) {
    //do something
        break
  }
}
```

Scopes

Scope is the place where variable is visible. And usually it's quite simple to understand, where new scope starts.

- { usually starts new scope, except classes.
- For statement generator starts new scope
- function definition starts new scope for parameters
- Every case clause starts new scope

Variables are not visible outside of their scope.

Names shadowing

Scopes are important, difference from Java for them is ”name shadowing”.

You can’t define new variable with the same name in the same scope, but you can do it in nested scope.

Function and closures

Function definition

Let's take a look to a function definition more attentively:

```
[annotations]
[modifiers] def name[Ts](param: Type)
                        (param2: Type)
                        (implicit param3: Type): ReturnType = {
  return expression
}
```

Different kinds of parameters

There are possibility to use following parameters type:

- Default parameters (param: Type = default)
- Repeated parameters (param: Type*). Only last parameter in every parameter clause. Actual type is Seq[Type].
- By-name parameters (param: =>Type). Actual type is () =>Type.
- On the call site you can use ”named parameters”. foo(paramName = expression)

Very probable situation to use named parameters to skip bunch of default parameters.

Local functions

Scala gives a possibility to use ”Local function” instead of private methods. Using this style you can get:

- Shorter parameter list because of possibility to use variables from enclosing function.
- Avoiding to trash scope of other public methods from this class, especially when you are using a lot of very small methods.

First-class functions

Syntax is very simple:

```
(param: Type, param2: Type) => resultExpression
```

As usual this syntax is syntactic sugar for creating new FunctionN class (while we are using it with Java 6, probably with Java 7 it will be changed to use invokeDynamic).

Placeholder syntax and other shorthands

There are possibilities to write functions shorter:

```
val x: Int => Int = t => t + 1
val y: Int => Int = _ => 1 //constant function
val z: Int => Int = _ + 1
val t: (Int, Int) => Int = _ + _
```

In compiler placeholder syntax is just syntactic sugar.

Partially applied functions

There are a lot of possibilities to write functions using existing definitions:

```
def sum(a: Int, b: Int): Int = a + b
val x = sum _
val y = sum(1, _)
List(1, 2, 3).foldLeft(0)(sum)
println(sum) //compile type error
```

Closures

Few words about closures. It's created on runtime. And all changes to variables are visible inside of closure.

Just be careful with NonLocalReturnControl inside of closure. Because outside of function nobody will catch it (remember that you will not have a stacktrace of this error).

Questions?