Theory: Branching

14 minutes 0 / 5 problems solved

Imagine that you want to write a function to get the absolute value: if integer argument `x` is negative, return `-x`, and otherwise, return the `x` itself. By definition, you will need something to separate the ways to get the result. The realization of such an approach in programming is called conditional execution, fork, or **branching**. Scala has the `if else` syntax for this process.

§1. If-else

The construction is quite intuitive: in a way, it resembles the way we speak when we set conditions. Take a look at this example:

```
scala> val a = -5
a: Int = -5
scala> if (a >= 0) print(a) else print(-a)
5
```

Here, we act conditionally on the value of `a`, and the condition has to be an expression with the `Boolean` result. The construction has two branches: if the condition is ***true,*** the part following the condition (**then-part**) is executed, and if otherwise, it's the part following else (**else-part**).

Note that `if else` is an expression, so it is evaluated and returns the result:

```
scala> val a = -5
a: Int = -5
scala> if (a == -5) -1 else 1
res1: Int = -1
```

The expression has to be consistent, so both ***then*** and ***else*** parts have to return the same type. If we don't follow this rule, we may receive results automatically resolved by the Scala compiler, so the result of the expression could be different from your expectations.

```
scala> if (a == -5) -1 else 1.0
res2: Double = -1.0
```

In the example above, we return `Int` or `Double`, and the compiler selects numeric with a wider area of values, which is `Double`.

You can omit the else-part, which returns *Unit*:

```
scala> if (a == -5) print(-a)
5
scala> if (a == -5) print(-a) else ()
5
```

§2. If-else-if

Sometimes the world is more complicated than left-right or black-white. So what do you do if your Scala program has more than two paths? For sure, you can make several `if else` constructions using nesting:

```
scala> val a = -5
a: Int = -5
scala> if (a < 0) print("negative") else
     | if (a % 2 == 0) print("even") else print("odd")
negative
```

Above, we added another branching to the else-part of expression. We can do the same for the then-part:

Current topic:

Branching

Topic depends on:

- ✕ Values and variables
- ✕ Functions introduction

Table of contents:

```
scala> if (a < 0)
     | if (a % 2 == 0) print("even") else print("odd") else
     | print("positive")
odd
```

Of course, you can add nesting in both parts without any limits on the nesting depth. Looks a little messy? Scala has additional syntax with `else-if` to make such expressions more readable:

```
scala> if (a < 0) {
     |   println("negative")
     | } else if (a % 2 == 0) {
     |   println("even")
     | } else {
     |   println("odd")
     | }
negative
```

If the condition is *false*, we have an additional check at `if else`, and only if this second condition is also *false*, we switch to the else-part. You can add as many `if else` sections as you want.

§3. One line vs multi-line

You've probably noticed in the previous examples that we define `if-else` constructions a little bit differently: one is one-line and without brackets, while the other has multiple lines. What option should you choose? The answer is simple: readability is the main point.

```
if (a == -5) -1 else 1

if (a > 0)
  println("The value of a is positive")
else if (a < 0)
  println("The value of a is negative")
else
  println("The value of a is 0")


if (a < 0) {

  println("The value of a is negative")

  if (a % 2 == 0)

    println("The value of a is even")

}
```

If you have short expressions in *then* and *else* parts, it's possible to place them all in one line. More complicated parts should be placed on multiple lines. If you have long multiline computations inside `if-else` , use brackets.

§4. Branching in functions

You can get the full potential of branching by defining it in functions. Note that `if-else` is an expression computed in place of definition. To use the expression several times, just define it as a function:

```
scala> def isEven(x: Int) = if (x % 2 == 0) true else false
isEven: (x: Int)Boolean
scala> isEven(10)
res3: Boolean = true
scala> isEven(13)
res4: Boolean = false
```

We can use conditional computation and call the function itself inside:

```
def count3(x: Int): Int =
  if (x == 0)
    0
  else if (x % 3 == 0)
    count3(x - 1) + 1
  else
    count3(x - 1)
```

Here, we introduced the function `count3` to count numbers $1...x$ that are a fraction of 3.

§5. Conclusion

Situations when we need to make a choice happen every day. That's why the ability to do things conditionally is one of the cornerstones of programming. In Scala, you can use `if-else` expressions to describe the logic with two branches. If you have more than two, feel free to use `if-else-if` systax. With these instruments, you can define the expression or function in different ways, but remember to take care of the readability of your programs.

Report a typo

11 users liked this theory. 0 didn't like it. What about you?

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion