Java → Scala^β → Functions introduction

Theory: Functions introduction

15 minutes 0 / 5 problems solved

Skip this topic Start practicing

181 users solved this topic. Latest completion was about 2 hours ago.

Current topic:

Functions introduction

Topic depends on:

Values and variables

Topic is required for:

Branching

Table of contents:

Intuitively, a **function** is a process that associates each element of a set X with a single element of a set Y, for example, `x = f(y)`. This is a mathematical point of view, but the same definition works for programming. In Scala, functions are essential for building programs: it is with these blocks that we can define the logic of our code.

§1. Functions in Scala

Functions are a set of instructions that are recomputed for every call. Typically, a function has a name and is defined with help of the **def** keyword:

```
scala> def f(x: Int) = x + 3
f: (x: Int)Int
scala> f(4)
res1: Int = 7
```

To call the function, use its name. A name of a function can be any valid identifier in Scala, which means that it is one or more characters and the sequence does not match any of the limited reserved combinations. This implies that you could potentially construct valid functions with interesting names:

```
scala> def +(i: Int ) = i + 1
$plus: (i: Int)Int
scala> def !#%&*+-/:<=>?@\^|~(i: Int ) = i + 1
$bang$hash$percent$amp$times$plus$minus$div$colon$less$eq$greater$qmark$at$bslash$up$bar$tilde: (i: Int)Int
scala> def #(i: Int ) = i + 1
<console>:1: error: identifier expected but '#' found.
       def #(i: Int ) = i + 1
           ^
```

As you see, the last definition in the example fails, because `#` is a reserved word.

§2. Arguments

For most functions, we want to use some input data. To define it, we should use **arguments** or **positional parameters**, so that every argument has its position in the function call:

```
scala> def g(x: Int, y: Character) = x - y
g: (x: Int, y: Character)Int
scala> g(100, 'a')
res2: Int = 3
scala> g('a', 3)
<console>:13: error: type mismatch;
 found   : Int(3)
 required: Character
       g('a', 3)
```

Every argument should have its place. If we have several, we can make a mistake. To avoid that, Scala supports another kind of syntax with argument names:

```
scala> g(x = 100, y = 'b')
res3: Int = 2
```

It is allowed to define functions without arguments:

```
scala> def currentMillis() = System.currentTimeMillis()
currentMillis: ()Long
scala> currentMillis()
res1: Long = 1583703616438
```

For the concluding group of arguments, you can specify a value to be used if the argument doesn't pass:

```
scala> def g(x: Int, y: Character = 'a') = x - y
g: (x: Int, y: Character)Int
scala> g(100)
res2: Int = 3
scala> g(100, 'b')
res3: Int = 2
```

Here, we force to use the character `a` if the argument `y` doesn't pass.

§3. Return type

In Scala, typically you don't have to specify the output type of a function: language compiler performs automatic detection of the result data type. This process is called **type inference**.

It is still recommended to specify the type if we can't easily understand the situation by reading the code. For example, for the function below, we explicitly inform to return numeric by adding `: Int` in the function definition:

```
scala> def g(x: Int, y: Character): Int = x - y
g: (x: Int, y: Character)Int
```

It is possible to define functions without output, which returns *Unit* type:

```
scala> def printInt(x: Int): Unit = println(x)
printInt: (x: Int)Unit
scala> printInt(5)
5
```

Scala has a reserved keyword `return` that you can use to return the result value for a function. However, a rule of thumb advises against using it, because this word can be easily omitted with no consequences:

```
scala> def bad(x: Int, y: Character): Int = return x - y
g: (x: Int, y: Character)Int
scala> def good(x: Int, y: Character): Int = x - y
good: (x: Int, y: Character)Int
```

§4. Conclusion

The main goal of our programs is transforming data. Functions are fundamental for any transformation; moreover, the function is a transformation itself. As Scala is a functional programming language, we can refer to functions as "first-class citizens": this reflects their central role in the programming process.

Report a typo

16 users liked this theory. 1 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion