Loops

Effective Programming in Scala

Loops

Most algorithms involve some kind of loops.

For instance, sorting a sequence requires iterating on its elements until they are sorted.

Finding the shortest path between two points in a graph requires iterating on the graph nodes and edges until the destination is reached.

And so on...

Implementing Loops

This lesson presents several ways of performing loops.

- iterating on the standard collections,
- *imperative loops* with the control structure `while`,
- *functional loops* with recursion.

Depending on the situation, some ways are more convenient than others.

Example: Factorial

The factorial of a positive integer `n` is the product of all positive integers less than or equal to `n`:

```
def factorial(n: Int): Int =
  1 * 2 * ... * (n - 2) * (n - 1) * n
```

Iterating on Collections

We have already seen that the Scala collections have operations that iterate on all their elements. The most general one is `foldLeft`. We can implement `factorial` with `foldLeft` as follows:

```
def factorial(n: Int): Int =
  (1 to n).foldLeft(1)((result, x) => result * x)
```

First, we create a collection of numbers from 1 to `n`, then we compute the product of all these numbers.

Iterating on Collections

We have already seen that the Scala collections have operations that iterate on all their elements. The most general one is `foldLeft`. We can implement `factorial` with `foldLeft` as follows:

```
def factorial(n: Int): Int =
  (1 to n).foldLeft(1)((result, x) => result * x)
```

First, we create a collection of numbers from 1 to `n`, then we compute the product of all these numbers.

Note: there is also an operation `product`:

```
def factorial(n: Int): Int = (1 to n).product
```

Aside: Ranges

`Range` is another type of Scala collection that models a range of values. Here are some self-explanatory examples:

```
1 to 4            // 1, 2, 3, 4
0 to 10 by 2      // 0, 2, 4, 6, 8, 10
5 until 0 by -1 // 5, 4, 3, 2, 1
```

The operations `to` and `until` are regular operations on type `Int`:

```
1.to(4) // : Range
```

Ranges have the operations of sequences, such as `map`, `filter`, etc.

Imperative Loops

Another way of doing loops consists in using the `while` control flow statement:

```
def factorial(n: Int): Int =
  var acc = 1
  var i = 1
  while i < n do
    i = i + 1
    acc = acc * i
  acc
```

`var` Definitions and `while` Loops

The keyword `var` introduces a variable definition. Unlike `val` definitions, `var` definitions can be **re-assigned** a new right-hand side over time.

The general syntax for the `while` control flow statement is:

```
while <condition> do <statement>
```

`var` Definitions and `while` Loops

The keyword `var` introduces a variable definition. Unlike `val` definitions, `var` definitions can be **re-assigned** a new right-hand side over time.

The general syntax for the `while` control flow statement is:

```
while <condition> do <statement>
```

Note: Scala 2 syntax was a bit different:

```
while (<condition>) { <statement> }
```

Functional Loops

Last, another way of doing loops is to use *recursive functions*. Recursive functions are functions defined in terms of themselves:

```
def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n - 1)
```

Here, `factorial(n)` is defined in terms of `factorial(n - 1)`, which is defined in terms of `factorial(n - 2)`, and so on until we reach `factorial(0)`, which terminates the loop.

Evaluating Recursive Functions

```
def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n - 1)
```

As an example, let's look at the evaluation steps of the expression `factorial(3)`:

- `3 * factorial(3 - 1)`
- `3 * 2 * factorial(2 - 1)`
- `3 * 2 * 1 * factorial(1 - 1)`
- `3 * 2 * 1 * 1`
- `6`

Summary

Looping can be achieved with several approaches:

- iterating on collections,
- `while` control-flow statements,
- recursive functions.

In the next lecture, we will see how these approaches compare in different situations.