This course has already ended.
The latest instance of the course can be found at: O1: 2023

Luet oppimateriaalin englanninkielistä versiota. Mainitut kuitenkin taustalykyisyä osaavasi suomea. Siksi **suosittelemme, että käytät suomenkielistä versiota**, joka on testatumpi ja hieman laajempi ja muokattuin mukaan.
Suomenkielinen materiaali kyllä esittelee englanninkielisetkin termit. Myös suomenkielisessä materiaalissa käytetään ohjelmien koodissa englanninkielistä nimiä kurssin alkupään johdonmukaisen esimerkkiajo kokonaisuutena.
Voit vaihtaa kielti A+ -n valikon yläreunassa olevasta painikkeesta. Tai tästä. **Vaihda suomeksi**.

Chapter 2.2: Inside an Object

About This Page

Questions Answered: How do I implement an (individual) object in Scala? What goes on inside an object when its method is called?

Topics: The program code of a singleton object: an object's variables; implementing methods; using `this` to refer to oneself.

What Will I Do? A small programming assignment follows a shortish read.

Rough Estimate of Workload? One hour or a bit more.

Points Available: A50.

Related Modules: IntroOOP.

Introduction

In this chapter, we'll use Scala to define the behavior of individual objects. As it turns out, that endeavor calls for skills that you already have. This is because an object definition has two parts:

- An object's methods are functions attached to the object. To define these functions, we can use techniques that are largely familiar from earlier chapters.
- An object needs to keep track of its attributes. For this purpose, we can use another familiar construct: variables.

The new thing here is how we connect functions and variables to an object.

Our Goal: An Employee Object

Let's create a singleton object that represents an individual employee in an otherwise imaginary accounting system. But first, let's see an example of how we intend to use the object.

An employee has attributes such as name and monthly salary; the code that defines the object defines the values of these attributes. We can request those values from the object:

```
employee.name=>s3: String = "Edelweiss Fume"
employee.monthlySalary=>s1: Double = 5000.0
```

The effect of method `raiseSalary` expects a number as a parameter. This number is a multiplier that is applied to the old salary. The following command, for instance, gives the employee a ten-percent raise:

```
employee.raiseSalary(1.1)
```

That command changed the object's state, as we can verify:

```
employee.monthlySalary=>s2: Double = 5500.0
```

Given a year as a parameter, the effect-free method `ageInYear` sends back a return value to let the caller know how old the employee will be in that year. The employee object knows its own year of birth and uses it to compute its age:

```
employee.ageInYear(2020)=>s3: Int = 55
```

The object has a working time, which is defined as a decimal number. The value 1.0 indicates a full-time worker (100-percent working time):

```
employee.workingTime=>s4: Double = 1.0
```

We can adjust the working time simply by assigning a new value to the attribute (as in the radio example of Chapter 2.1), thereby sending the object a message: "Set your working time to 0.6." The value 0.6 indicates part-time work; we'll make our example employee work sixty percent of a full week:

```
employee.workingTime = 0.6employee.workingTime: Double = 0.6
```

The effect-full `monthlyCost` method determines the employee's monthly cost to their employer. Our object computes this cost as the product of the monthly salary (now 5,500 euros per month), the working time (60%), and a multiplier to cover additional costs (a parameter; 1.3 in the example below):

```
employee.monthlyCost(1.3)=>s5: Double = 4290.0
```

Finally, the `description` method returns a string that identifies many characteristics in English. This effect-free method takes no parameters, so we don't need any brackets as we call it (just as we didn't when we asked the object for its name and salary above). Here, we ask the object for its description and then print what we get:

```
println(employee.description)=>Edelweiss Fume (b. 1965), salary 0.6 * 5500.0 e/month
```

Defining an Object and Methods in Code

The employee object: how it looks on the inside

Here's the code that defines our employee object. You can also find the code in `singletons.scala` within the IntroOOP module.

Let's begin with an overview before we look at each method in detail.

```
object employee {

  var name = "Edelweiss Fume"
  val yearOfBirth = 1965
  var monthlySalary = 5000.0
  var workingTime = 1.0

  def ageInYear(year: Int) = year - this.yearOfBirth

  def monthlyCost(multiplier: Double) = this.monthlySalary * this.workingTime * multiplier

  def raiseSalary(multiplier: Double) = {
    this.monthlySalary = this.monthlySalary * multiplier
  }

  def description =
    this.name + " (b. " + this.yearOfBirth + "), salary " + this.workingTime + " * " + this.monthlySalary + " e/month"

}
```

The definition of a singleton object begins with the word `object`. The rest word is a programmer-chosen name for defining it.
The body of the definition goes inside curly brackets. The brackets are required by the Scala language.
Within the brackets, we have the object's variables; their definitions look familiar. This is where we assign (initial) values to the variables. The value of an object's `val` variable can only be examined (as in `employee.yearOfBirth`), but a `var` variable can be assigned a new value, as we did with the employee's working time in the REPL.
Below, we have the object's methods, whose definitions start with the word `def`. There is no rule in Scala that forces us to write the variables above the methods, but the programmers prefer to do so. More on the methods below.
Notice the indentations that increase in size to highlight the program's structure.

The employee object: how it works on the inside

The employee's methods detailed

```
def ageInYear(year: Int) = year - this.yearOfBirth

def monthlyCost(multiplier: Double) = this.monthlySalary * this.workingTime * multiplier

def raiseSalary(multiplier: Double) = {
  this.monthlySalary = this.monthlySalary * multiplier
}

def description =
  this.name + " (b. " + this.yearOfBirth + "), salary " + this.workingTime + " * " + this.monthlySalary + " e/month"
```

Scala's keyword `this` means "the object whose method is being executed". Or, from the object's perspective: "I myself". Follow the word with a dot, and you can refer to different members of the object. As shown in the preceding animation, `this` is effectively a parameter variable that works just like other parameter variables: it receives its value from the left of the dot in each method call.
The code instructs the object: "When your method `ageInYear` is called, respond with a number that you get by subtracting the value of your own `yearOfBirth` variable from the given year."
"Multiply your own monthly salary, your own working time, and the given multiplier (the last of which is stored in a local parameter variable during the method call)."
"Multiply your own monthly salary with the given multiplier and assign the result to be your monthly salary (replacing the old value)."
Chapter 1.7 introduced some punctuation rules and conventions for defining functions. They apply to methods, too. It's common to omit the curly brackets from an effect-free method if the method body is just a single expression.
`description` takes no parameters and doesn't even have a parameter list.
On `this`:

In O1, we'll use the word `this` pretty much every time we wish to refer to the members (variables and methods) of the object that is executing a method. In a technical sense, however, there is often no strict requirement to use the word as we do.

Sometimes it is necessary to use `this`. As an example, below is a different version of the employee object that differs from our earlier version in just the name of a single variable:

```
object employee {

  var name = "Edelweiss Fume"
  val year = 1965
  var monthlySalary = 5000.0
  var workingTime = 1.0

  def ageInYear(year: Int) = year - this.year
  // etc.
```

In this version, the year of birth is stored in a variable called simply `year`, not `yearOfBirth`, as in the original.
Therefore, the object's variable has the same name as `ageInYear`'s parameter variable.
The name `year` without a preceding `this` means the year given as a parameter.
The word `this` specifies that we mean the object's variable, not the parameter. If we were to omit it from this version, the method would subtract its parameter value from that very same value, and would therefore always return zero.

In our original employee object, it would have been possible to omit each `this` prefix, because none of the local variables had the same name as the object's variables.

Even when it is not required, the word `this` enhances the programmer's view an object's variables, as opposed to where they use local variables. We recommend that you always use `this` when referring to an object's own variables, because it makes the code easier to understand.

To an extent, this is a matter of taste. If you wish — and if you know what you're doing — go ahead and leave out any `this` keywords that don't need to be there. Outside of O1, such omissions are common.

Create an Object

Next, you'll get to write an object that represents a bank account. The object should work as shown in the following example.

The account object: an example scenario

An account has a balance (in euro cents) and an account number. We can ask the object to report this information:

```
account_balance=>s1: Int = 0
account_number=>s2: String = "2150300000077U7100
```

We can deposit money in the account. Here, we choose to represent sums of money as integers that correspond to euro cents. Let's deposit 200 euros and review the balance:

```
account.deposit(20000)account_balance=>s3: Int = 20000
```

Depositing a negative sum doesn't affect the balance:

```
account.deposit(-1000)account_balance=>s3: Int = 20000
```

The `withdraw` method takes money from the account and returns the amount that was successfully withdrawn:

```
account.withdraw(30000)res10: Int = 5000
```

Withdrawals that would result in a negative balance aren't allowed. Below, we get only 150.00 euros as we empty the account:

```
account.withdraw(30000)res11: Int = 15000
account_balance=>s12: Int = 0
```

Your assignment

Write an object `account` that works as described above. To summarize the main points:

- It has a number "`2150300000077U7100`".
- It has a balance that is initially zero but may change.
- It has a method `deposit` that adds a given amount of money to the account (as long as the given parameter value is positive) and doesn't return anything.
- It has a method `withdraw` that reduces the account's balance by a given amount, or empties the account in case the given parameter is greater than the current balance. The method returns the amount that was actually withdrawn.
  - In this assignment, you're not required to consider the possibility that a negative number is passed to `withdraw`. But if you want, you can write the method to leave the account untouched in such cases.

Write your code within the file `skiingHuman.scala` in IntroOOP. There's a comment in the file that shows where exactly.

How to go about it

We suggest the following approach:

1. Study the example scenario in detail. *Note the names and types of each variable!*
2. Find the indicated spot in `singletons.scala`. The first line of the object definition is already there, as are the required curly brackets.
3. Define the object's variables inside the curlies by assigning values to them. Indent your code with two space characters per line.
4. See if it works. Fire up your REPL in the IntroOOP module and try the expressions `account_balance` and `account_number`.
5. Write the methods `deposit` and `withdraw`.
   - Use the `min` and `max` functions known to you from earlier chapters.
   - The algorithm you need for `withdraw` should be already familiar from a program in Chapter 1.8.
6. Reset the REPL session and test your methods on different parameter values. (Reminder: you may find it convenient to use the Rerun button in the REPL's top-left corner.)
7. Submit your code only when you're convinced that your object works as specified.

Points A **68 / 50** My submissions **4 / 50** ▾ Deadline Wednesday, 23 September 2020, 12:00
To be submitted alone or in groups of 2

This course has been archived (Tuesday, 31 August 2021, 23:59).

Assignment 4 (IntroOOP: account)

Select your files for grading

singletons.scala

Choose File | No file chosen

Submit

Something wrong?

Once you've done with the assignment, think about the object you just defined.

Did you define a `var` variable named `balance` for your account object? If you did, that's all good. But there's something a bit questionable about this variable that may have already occurred to you.

We took it upon ourselves to create an account object whose balance is modified through `deposit` and `withdraw`, which apply due diligence and prevent the balance from becoming negative. However, there is presently nothing that prevents us from commanding the object to set its balance to a negative value like this: `account.balance = -100000`

Obviously, this is no problem at all if nobody ever issues such a command. We could say that assigning a negative balance counts as an error on the part of whoever uses the account object. However, there are many programmers who agree that programs should be written so that they minimize the risk of such mistakes. Soon enough, in Chapter 3.2, you'll learn how to restrict the operations on an object's variables. Right now, you don't need to worry about the matter.

There are styles of programming where it wouldn't be considered problematic to make an inappropriate command available. Scala programmers, however, tend to be among those whose who recommend *minimizing* such opportunities for error so as to be more efficient and build more reliable software.

Summary of Key Points

- Some of the objects in Scala programs are so-called singleton objects: objects defined on an individual basis.
- The definition of a singleton object comprises variable definitions and method definitions.
  - You can define a variable to be part of an object. Such a variable can be assigned values just as we've done with other variables.
  - You can define methods for an object just as we've defined functions before.
- It is extremely common that a method needs to use one or more of the object's own variables or other methods. You can use Scala's keyword `this` to make an object refer to itself and its own components.
- Links to the glossary: object, singleton object, `this`.

An updated concept map:

Feedback

Please note that this section must be completed individually. Even if you worked on this chapter with a pair, each of you should submit the form separately.

Accepted My submissions 1 ▾

This course has been archived (Tuesday, 31 August 2021, 23:59).

Time spent: (*) Required

Please estimate the total number of minutes you spent on this chapter (reading, assignments, etc.). You don't have to be exact, but if you can produce an estimate to within 15 minutes or half an hour, that would be great.

60

"I feel that I have understood the most important things in this chapter." (*) Required

- fully agree
- somewhat agree
- somewhat disagree
- fully disagree
- I'm unable to answer or don't want to comment.

Written comment or question:

You aren't required to give written feedback. Nevertheless, please do ask something, give feedback, or reflect on your learning! (However, the right place to ask urgent questions about programs that you're currently working on isn't this form but Piazza or the lab sessions. We can't guarantee that anyone will even see anything you type here before the weekly deadline.)

Submit an update

Credits

Thousands of students have given feedback that has contributed to this ebook's design. Thank you!

The ebook's chapters, programming assignments, and weekly bulletins have been written in Finnish and translated into English by Juha Sorva.

The appendices (glossary, Scale reference, FAQ, etc.) are by Juha Sorva unless otherwise specified on the page.

The automatic assessment of the assignments has been developed by (in alphabetical order) Riku Autio, Nikolas Drosdek, Joonatan Honkamaa, Jaakko Kantojärvi, Niklas Kröger, Teemu Lehtinen, Strasdosky Otewa, Tim Sarjakoski, Teemu Sirkiä, and Aleksi Vartiainen.

The illustrations at the top of each chapter, and the similar drawings elsewhere in the ebook, are the work of Christina Lassheikki.

The animations that detail the execution Scala programs have been designed by Juha Sorva and Teemu Sirkiä. Teemu Sirkiä and Riku Autio did the technical implementation, relying on Teemu's Jsvee and Kelmu toolkits.

The other diagrams and interactive presentations in the ebook are by Juha Sorva.

The O1:library software has been developed by Aleksi Lukkarinen and Juha Sorva. Several of its key components are built upon Aleksi's SMCL library.

The pedagogy of using O1:library for simple graphical programming (such as `Pic`) is inspired by the textbooks *How to Design Programs* by Flatt, Felleisen, Findler, and Krishnamurthi and *Picturing Programs* by Stephen Bloch.

The course platform A+ was originally created at Aalto's LeTech research group as a student project. The open-source project is now hosted by the department's edu-tech team and hosted by the department's IT services. Markku Riekkinen is the current lead developer; dozens of Aalto students and others have also contributed.

The A+ Courses plugin, which supports A+ and O1 in IntelliJ IDEA, is another open-source project. It was created by Nikolai Denisov, Olli Kilkanen, and Nikolas Drosdek with input from Juha Sorva, Otto Seppälä, Arto Hellas, and others.

For O1's current teaching staff, please see Chapter 1.5.

Additional credits appear at the end of some chapters.