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

« Week 11 Course materials Chapter 11.2: Robots That Compete »
CS-A1110 / Week 11 / Chapter 11.1: Arrays and a Faulty Train

Luet oppimateriaalin englanninkielistä versiota. Mainitsit kuitenkin taustakyselyssä osaavasi suomea. Siksi **suosittelemme, että käytät suomenkielistä versiota**, joka on testatumpi ja hieman laajempi ja muutenkin mukava.
Suomenkielinen materiaali kyllä esittelee englanninkielisetkin termit. Myös suomenkielisessä materiaalissa käytetään ohjelmien koodissa englanninkielisiä nimiä kurssin alkupään johdantoesimerkkejä lukuunottamatta.
Voit vaihtaa kieltä A+:n vaikon yläreunasta olevasta painikkeesta. Tai tästä. **Vaihda suomeksi**.

Chapter 11.1: Arrays and a Faulty Train

About This Page

Questions Answered: How about some more practice on loops, code-reading, and debugging? The word "array" has appeared here and there — what does it mean?

Topics: See above.

What Will I Do? Start by reading a bit. Then spend most of the chapter reading and debugging a given program.

Rough Estimate of Workload: Three or four hours? Remember to ask for help if you get stuck in the debugging assignment.

Points Available: 880.

Related Modules: Train **(new)**.

Introduction: What's an Array?

You've already seen the word "array" in O1:

```
val numbers = Buffer(1, 2, 3)numbers: Buffer[Int] = ArrayBuffer(1, 2, 3)
val words = "one,two,three".split(",")words: Array[String] = Array(one, two, three)
```

`ArrayBuffer` appears in the REPL output when we use buffers.
The `split` method (Chapter 5.2) does a thing and returns the pieces — not in a vector or a buffer but an array.

Arrays (taulukko) are collections of elements:

- Like buffers and vectors, arrays store elements at specific indices.
- You can replace an array element with another. In this respect, an array is similar to a buffer and different from a vector.
- However, the size of an array is fixed, just like a vector's is. The size is set when the array is created and the number of indices in an array never changes.

A very familiar sort of construct, then. Let's now look at a few examples of using an array before we discuss why you might want to use them.

Arrays in Scala

The `Array` type

Arrays, like vectors and unlike buffers, are always available in Scala programs without an `import`.

Creating an array looks familiar:

```
val myArray = Array("first", "second", "third", "fourth")myArray: Array[String] = Array(first, second, third, fourth)
val anotherArray = Array.tabulate(5)( index => 2 * index )anotherArray: Array[Int] = Array(0, 2, 4, 6, 8)
```

Indices and methods also work like you'd expect:

```
myArray(2)res0: String = third
myArray(3) = "last"myArray:myArray: Array[String] = Array("first", "second", "third", "last")
myArray.sizeres1: Int = 4
myArray.indexOf("third")res2: Int = 2
myArray.mkString(":")res3: String = first:second:third:last
myArray.map(_.length )res4: Array[Int] = Array(5, 4, 6, 4)
```

Appending an new element and thereby increasing the size of the array is impossible:

```
myArray += "one more?"<console>:13: error: type mismatch;
```

Creating an unitialized array with `ofDim`

Sometimes, it's convenient to create a collection of a specific size whose contents are initialized only later. For that, you can use a method that isn't available for vectors or buffers, `Array.ofDim`:

```
val myArray = Array.ofDim[Int](5)myArray: Array[Int] = Array(0, 0, 0, 0, 0)
```

`ofDim` needs a type parameter. Here, we have elements of type `Int`.
As a regular parameter, we pass in a number that sets the array's size. Here, we create an array with five elements.
The method creates and returns an array of the specified size. It contains copies of some default value; here, that value is zero.
Creating an array so reserves the desired number of memory slots for the actual array elements, which we haven't really set yet but whose (maximum) number we know.

You can read `Array.ofDim` as "array of dimensions". Like that name implies, the method works for creating "multidimensional" (nested) arrays. They are similar to the "multidimensional" vectors of Chapter 6.1:

```
val twoDimensional = Array.ofDim[Int](2, 3)twoDimensional: Array[Array[Int]] = Array(Array(0, 0, 0), Array(0, 0, 0))
val threeDimensional = Array.ofDim[Double](2, 2, 2)threeDimensional: Array[Array[Array[Double]]] =
Array(Array(Array(0.0, 0.0), Array(0.0, 0.0)), Array(Array(0.0, 0.0), Array(0.0, 0.0)))
```

Note: `ofDim` just creates the array, it doesn't put any meaningful content in it. Still, any array you create like this always contains *something*. The default value depends on the element type:

- for numerical types (`Char` included), it's zero;
- for Booleans, it's `false`; and
- for all other types — including `String`, for instance — it's the "non-existent value" `null` (which is a spawning bed for errors; Chapter 4.3).

You can and almost always should replace the default values with something more meaningful sooner or later.

Careful with those default values!

Suppose you have a class `Footballer` and you create this array:

```
val finnishTeam = Array.ofDim[Footballer](11)
```

It's a common beginner's error to forget that the above command does not actually create even a single instance of the `Footballer` class, even though it ostensibly creates "an array of footballers". What you get is an array that contains eleven copies of the `null` reference. If you want to create `Footballer`s and store them in the array, you'll need to do that separately:

```
finnishTeam(0) = new Footballer("Tirja-Riikka Korpela")
finnishTeam(1) = // etc.
```

Unless there is an actual object reference at the appropriate index, commands like these will result in a `NullPointerException` at runtime:

```
finnishTeam(9).score()
val keeper = finnishTeam(0)
println(keeper.name)
```

Doesn't Sound Too Useful?

Since an array's size never changes, the `Array` type supports fewer operations than the `Buffer` type; any piece of functionality that you might wish to implement using arrays you can also implement using buffers. (There reverse is also true.) If you want an immutable, numerically indexed collection, you can use a vector. Uninitialized arrays are infested with `null`s.

Given all that, why bother with yet another collection type? Here are a few reasons:

- **Commonness:** Arrays are part of a programmer's general knowledge. They are a basic data structure that is used for implementing other collection types. Arrays are available in many programming languages; in quite a few of languages, they are the most common type of collection.
- **Efficiency:** Arrays sometimes make a program more efficient, which is due to differences in the implementations of the various collection types. As far as Scala arrays are concerned, this is one of the more likely reasons why you might occasionally opt for an array, but — once again — we leave efficiency concerns for later courses to deal with. The Scala website has a table that compares the efficiency characteristics of different collections.
- **Mutable image scenarios:** An array is an intuitive choice when you need a collection whose contents can change but whose size cannot. The `Grid` class of Chapter 7.4, for instance, has been implemented using a "two-dimensional" array. You can also consider an array if you need a collection whose size is capped at a preset limit.
- **Incidence in libraries:** The `Array` type crops up in some software libraries, including Scala's core API. The aforementioned `split` method is an example.

Arrays as an implementation tool

The word `ArrayBuffer` reveals how Scala's buffer class has been implemented: each buffer object internally stores its elements in an array of some fixed size. When that array runs out of capacity, the buffer object swaps it for a bigger array where it copies the old array contents and adds any new ones.

When you use such a buffer you therefore also use arrays, albeit indirectly and nearly unnoticeably.

Arrays have also been used (differently) for implementing Scala's `Vector` class.

`ArraySeq`: an immutable array

Yet another collection type in the Scala API is called `ArraySeq`. An `ArraySeq` is similar to an array (and implemented using one), except that it doesn't provide no effectful methods. An `ArraySeq` is thus immutable in principle; it resembles a `Vector` but differs from `Vector`s in its efficiency characteristics.

```
import scala.collection.mutable.ArraySeqimport scala.collection.mutable.ArraySeq
val unchangingArray = ArraySeq(10, 4, 5)unchangingArray: ArraySeq[Int] = ArraySeq(10, 4, 5)
unchangingArray(2)res5: Int = 4
unchangingArray(1) = 5 *
  error: value update is not a member of ArraySeq[Int]
```

Debugging a Train

This assignment puts you in a position that is familiar to many professional programmers: you need to untangle a mess of code that someone else wrote.

Task description

The `Train` module contains classes that represent train cars and the seats and cabins in those cars. Think of the classes as (poor) parts of an imaginary software system where customers can reserve places on a train. The code is written in a heavily imperative style: it is built on arrays, `do` and `while` loops, and effects on mutable state.

The code isn't even close to exemplary. The worst thing is that it doesn't work right.

Your task is to write an app object that tests the given classes and to use it to locate the errors in the program. You should also fix the errors, but that's the easy part.

See the Scaladocs for how the classes should work.

Instructions and hints

- Please don't get stuck on this assignment. Use the lab sessions and the online forums for hints. There's a lot more to do in Week 11 after this!
- First, *test* the code thoroughly. Call the methods on different values and in different order. Work out what works and what doesn't work. Then debug the code: find and fix what causes the problems.
- Write an app object for testing. You may also wish to use IntelliJ's debugger.
  Without the debugger, this assignment may be significantly harder than it needs to be.
- There are eight bugs. Each of them is in a distinct location in the program code.
- Various methods work peculiarly or crash the program if you pass in 'obviously silly' values (e.g., if you set a negative number of cabins or if you add a `null` to the train instead of an actual train-car). In this assignment, that does not count as a bug. Concentrate your efforts on finding behaviors that clearly contradict the specification even on valid parameters.
- `SittingCar` is the most complex of the classes. Inspect the other classes first. Look at `SittingCar` last, when you'll have developed a better understanding of the given code and a better workflow for testing.
- Some of the classes have companion objects in the same file. Those objects are there just to store constants.
- You might feel the urge to improve the programming style or efficiency of the given code. You're not required to do that, though.
- So-called **desk checking** (pöytätestaus) can help, as suggested by a former O1 student:
  > In its own way, this was a tough assignment, but in the end I solved it pretty easily once I printed out the code, picked up my pencil, and took the whole stack of papers with me for reading in the sauna. :)
  >
  > Sauna is probably inessential in this method.

Points 8 `0 / 80` My submissions `0 / 10` Deadline Wednesday, 2 December 2020, 12:00
To be submitted alone or in groups of 2

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

Assignment 1 (Train)

Select your files for grading

SittingCar.scala
Choose File | No file chosen

SleepingCar.scala
Choose File | No file chosen

Train.scala
Choose File | No file chosen

TrainCar.scala
Choose File | No file chosen

Submit

Summary of Key Points

- Arrays are a mutable collection of fixed size.
  - Arrays are common in many programming languages and programs.
  - Depending on the context, using arrays instead of other collections may bring benefits such as improved efficiency.
- Links to the glossary: array; testing.

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.

180

"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, Scala 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, Timi Seppälä, 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 diagrams and interactive presentations in the ebook are by Juha Sorva.

The O1Library 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 O1Library 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 shepherded by the Computer Science 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 Denissov, Olli Kiljunen, and Nikolas Drosdek with input from Juha Sorva, Otto Seppälä, Arto Hellas, and others.

For O1's current teaching staff, please see Chapter 1.1.

Additional credits appear at the ends of some chapters.