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

« Chapter 7.3: Inheritance and Class Hierarchies Course materials Chapter 7.5: City Simulator »
CS-A1110 / Week 7 / Chapter 7.4: A Game of Glasses

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 kielti A+:n valikon yläreunassa olevasta painikkeesta. Tai tästä. **Vaihda suomeksi**.

Chapter 7.4: A Game of Glasses

About This Page

Questions Answered: How will I manage with a bigger program with multiple classes?

Topics: The chapter revolves around a single programming assignment, which features inheritance and collection methods, among other things.

What Will I Do? Study given code and program.

Rough Estimate of Workload? Two or three hours.

Points Available: 860.

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

The Game of Viinaharava


A game of Viinaharava in progress. Some of the glasses have been drunk; those squares either contain a number indicating the number of neighboring booze glasses or are empty to indicate that there is no booze in the immediate vicinity. (Yes, it's a different-looking Minesweeper.)

The local temperance society has commissioned a game that promotes water as a healthy drink. To that end, a game named Viinaharava has been designed; its implementation is more or less ready but needs you to flesh it out.

Viinaharava takes place on a board that consists of small drinking glasses arranged in a grid. Most of them contain water but a few contain a stiff, transparent alcoholic drink: "a booze". The player's task is to drink all the water glasses without touching a booze.

The player virtually drinks a glass by clicking on it. Their task is simplified by the fact that there's a hint at the bottom of each glass: the number of boozes in neighboring glasses. The game is over when either all the water or even a single booze has been drunk.

Task description

You'll find a partially operational implementation in the Viinaharava module. See below for an introduction.

Study this module and fill in the missing parts. You may wish to follow these steps:

1. Launch Viinaharava with the app object `o1.viinaharava.gui.Viinaharava`. Notice: the board shows up but the game doesn't work.
2. Study the class `o1.Grid`, which has been used in Viinaharava's implementation. See below for further information.
3. Familiarize yourself with the classes in package `o1.viinaharava`. Start from the overview of the package below, then turn to the Scaladocs and the source code.
4. Once you understand the program as given, add the missing parts. See below for additional instructions and hints.

Representing Dense Grids

You'll remember Snake from Chapter 6.3. In that game, the snake and its food were located on the spaces of a grid-like playing field, which we recorded as `GridPos` objects. Each `GridPos` was composed of two integers `x` and `y`, the pair of which pinpointed a space on the grid.

Viinaharava resembles Snake: it, too, has a playing field that is essentially a grid. We can again use `GridPos` as we represent the locations of each glass on the game board.

(In this assignment, you don't have to concern yourself with pixels or graphics. The given GUI takes care of that. You can focus on modeling the rules of the game itself. It suffices to consider each location in terms of its position on the grid, a `GridPos`.)

In Snake, we had a "sparse" grid: there were few actual items (snake segments; food) in the grid compared to the total number of spaces. We represented the game's state by simply tracking those `GridPos` coordinates that actually did contain something and considered each other space to be empty.

This time we'll be different and represent game boards as "dense" grids. We'll record, for every single space on the board, which kind of glass it contains: Is it a water glass or a booze? Have the contents been drunk already? How many dangerous neighbors does it have?

We'll find it easier to represent dense grids if we adopt a tool designed for just that purpose, class `Grid`.

Class `Grid`

The `o1` package provides a `Grid` class. Each `Grid` instance represents a grid that consists of elements of similar size that have been laid out in rows and columns; the elements could be glasses, for instance.

The class has a number of methods for manipulating such grids. For instance, there are methods for picking out a particular element given its position (`elementAt` and `apply`), finding all the spaces that are adjacent to a given space (`neighbors`), and determining the grid's dimensions (`width`, `height`, and `size`).

`Grid` is an abstract class. We can't simply call `new Grid`; we need to instantiate it via a subclass. The abstract `Grid` class is designed to work in different applications that feature grids and `GridPos`es: it doesn't specify what kind of spaces grids consist of. That's something we'll need to specify in a subclass.

Viinaharava is a particular use case for `Grid`: each game board is a grid that consists of objects that represent glasses. (In later assignments, we'll use `Grid` to represent grids with other kinds of content.)

The Viinaharava Module

Module Viinaharava contains two packages. We won't go into the GUI package `o1.viinaharava.gui` and you don't need to understand how it works: it's enough that you find the app object there and use it to start the program. The parent package `o1.viinaharava`, however, is very topical. Its two key classes are:

- `Glass`: instances of this class represent individual glasses that the game board consists of.
- `GameBoard`, a subclass of `Grid`: a `GameBoard` object represents an entire game board, a grid of `Glass`es.

The diagram below describes the relationships between the classes:


The lower part of the diagram means that each game board is associated with multiple glasses, each at its particular position: we can use a `GridPos` to pick out a particular `Glass` on a `GameBoard`.

`Glass` and its missing methods

Each glass can be either full or empty. It can be either a glass of water or a glass of booze. Moreover, each `Glass` object keeps track of how dangerous it is: how many boozes there are in the adjacent glasses. The danger level is a number between zero and eight; diagonally adjacent counts, too.

`Glass` objects have instance variables for recording their contents and danger level. Each glass also "knows" which game board it's on and which `GridPos` it's at.

When created, a glass is full of water. The `Glass` class has methods for modifying that initial state. Specifically:

- We can empty a glass. The `empty` method is invoked whenever the user (left-)clicks a glass in the GUI.
- We should be able to fill a glass with booze (`pourBooze`). This has the additional effect of increasing the danger levels of neighboring glasses. `pourBooze` is called several times at the start of each game to place the hidden booze on the board. (For testing purposes, the GUI also lets the player add booze during a game.)

`pourBooze` lacks an implementations, though. The `neighbors` method, which is supposed to find the adjacent glasses, is also missing.

`GameBoard` and its missing methods

Here's a start for the `GameBoard` class:

```
class GameBoard(width: Int, height: Int, boozeCount: Int) extends Grid(width, height) {
  // ...
}
```

A new `GameBoard` instance needs three constructor parameters: the number of columns on the grid, the number of rows, and the number of booze glasses initially hidden on the board.
Initializing any `Grid` object requires a width and a height. We pass these two parameters on to the superclass.

The class header needs one more thing before it works. This is because the superclass `Grid` demands a type parameter in addition to the constructor parameters. Just like we have used square brackets to mark the element type of a `Buffer`, we can mark the element type of a `Grid`:

```
class GameBoard[width: Int, height: Int, boozeCount: Int) extends Grid[Glass](width, height) {
  // ...
}
```

A `GameBoard` object is a `Grid` whose elements are `Glass` objects.

As you saw when you launched the game, the given implementation already fills the board with water glasses. A further inspection of the given code in `GameBoard.scala` shows us how:

```
class GameBoard[width: Int, height: Int, boozeCount: Int) extends Grid[Glass](width, height) {

  def initialElements = {
    val allLocations = (0 until this.size).map( n => GridPos(n % this.width, n / this.width) )
    allLocations.map( loc => new Glass(this, loc) )
  }

  this.placeBoozeAtRandom(boozeCount)
```

As the documentation says; this method, which produces a collection of all the elements that initially occupy the grid, is left as abstract by the superclass `Grid`. (However, the superclass automatically calls this method when a new `Grid` is created.)
The subclass `GameBoard` implements the method by returning a collection of empty `Glass`es. Feel free to study this implementation, but it's not strictly necessary for the present assignment. Don't change this method.
The `placeBoozeAtRandom` call written directly into the class body is part of the code that initializes new instances of `GameBoard` (i.e., the class's constructor). The method is invoked every time a new `GameBoard` is created.
The aforementioned `placeBoozeAtRandom` method doesn't have an implementation yet, so there's no booze on the board. That will require your attention.

The `drink` method is also missing, which is why the game doesn't do anything when clicked. So's `isOutOfWater`, which the app uses for determining when the game is over.

You may tackle with the assignment in three steps as described below.

Recommended Workflow

Step 1 of 3: water

In `GameBoard.scala`, find the `drink` method and write the missing `if` branch that deals with water glasses.

Then implement `isOutOfWater` in the same class.

- For easy access to all the glasses on the game board, you can use the `allElements` method that `GameBoard` inherits from `Grid`.
- If you pick the right higher-order method (from Chapter 6.3), the implementation will be quite simple.

Try running the game again. You can now empty glasses to your heart's content. Once all the water is gone, the app lets you know. The game still lacks the booze and the consequent suspense.

Step 2 of 3: placing the booze

Implement `neighbors` on `Glass`. Hint: use an existing method for a very simple solution.

Then write the `pourBooze` method in the same class. Once that's done, it's possible pour booze in glasses and thereby adjust the danger levels of neighboring glasses. The actual game still works as before, however, since the newly implemented method doesn't get called.

Switch your attention to `placeBoozeAtRandom` in class `GameBoard`, a private method. Implement this method so that it selects a random set of glasses and pours booze in them. The method should randomize the glasses in such a way that each new game (each new `GameBoard`) is unpredictable.

Here are two different ways to approach the problem. Feel free to pick either of them, or come up with something else, as long as your method works.

Algorithm #1:

1. Use a random-number generator to pick a pair of coordinates.
2. Find out if those coordinates already contain booze.
   - If so, do nothing.
   - If not, pour booze there.
3. Keep repeating steps 1 and 2 until the target number of booze glasses is reached.

Which one is better?

If you want, you can reflect on which of these two algorithms demands more work (time) from the computer. How does the amount of work depend on how big the game board is and how many booze glasses you intend to place?

Algorithm #2:

1. Form a collection that contains each of the glass objects.
2. Shuffle the collection so that the glasses are in random order. (It's possible to write, say, a loop that does this, but you can also use the convenient method `Random.shuffle`; see below for an example.)
3. Take the desired number of glasses from the collection. Pour booze in each of those glasses.

Here's an example of `shuffle`:

```
import scala.util.Random; import scala.util.Random
val numbers = (1 to 10).toVectornumbers: Vector[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Random.shuffle(numbers)res0: Vector[Int] = Vector(8, 9, 7, 4, 6, 1, 10, 2, 5, 3)
Random.shuffle(numbers)res1: Vector[Int] = Vector(8, 9, 4, 5, 9, 1, 3, 7, 2, 10)
```

Step 3 of 3: drinking

Try running the program again. It should be more or less playable now, but let's make one more change.

When the player hits a booze glass and the game is over, we'd like the game to reveal (i.e., empty) all the booze glasses on the board.

Write the branch of the `drink` method that deals with booze glasses. Make use of `boozeGlasses` in class `GameBoard`, which returns all the booze glasses on the board in a vector.

Try the program.

One drink per click, please

When the player clicks on a water glass and reveals that it had a danger level of zero, one can safely drink all the neighboring glasses as well. Perhaps you'd like the program to do so automatically without the player having to click on each safe neighbor separately. In Chapter 12.1 we'll do just that with the help of a technique known as recursion.

Points 8 (60 / 60) My submissions 1 / 10 ▾ Deadline Wednesday, 28 October 2020, 12:00
To be submitted alone or in groups of 2.

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

Assignment 20 (Viinaharava)

Select your files for grading

GameBoard.scala
Choose File | No file chosen

Glass.scala
Choose File | No file chosen

Submit

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

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 Lasshekki.

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 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.