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

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 valikon yläreunassa olevasta painikkeesta. Tai tästä: **Vaihda suomeksi**.

Chapter 7.5: City Simulator

About This Page

Questions Answered: How about some more practice? How about an example of modeling real-world phenomena in a program?

Topics: The main themes are (still) inheritance and collection methods. Algorithm-writing practice. The optional assignment is an example of applying inheritance while refactoring code.

What Will I Do? Study given code and program.

Rough Estimate of Workload:? Three or four hours?

Points Available: C60.

Related Modules: CitySim **(new)**. AuctionHouse2 **(new)** in the optional assignment.

../_images/person04.png

Citizens in a Simulator

> ../_images/schelling1.png
> Our simulator is based on a grid that represents a city map. Each location on the grid is an address that a citizen (or family) can move into. Blue and red represent different demographics. Addresses marked in white are currently vacant.

One of the most powerful things about programming is that you can create dynamic models of phenomena and processes in the real world. In this assignment, we'll form a computational model of a social phenomenon. You'll take control of an application that simulates people's movements on a city map; in particular, this simulation will explore how demographics may impact on the social layout of a city.

In this context, a *demographic* is any subset of the city's population that the citizens may perceive to be relevant as they assess their own neighborhood. For instance, we can split the citizens in demographics on the basis of their financial status, political views, ethnic background, or age. In this chapter, our simulator will have only two demographics, red and blue. In Chapter 9.2, we'll extend the simulator to support more demographics.

At the start of each simulation run, we place red and blue citizens at random addresses on the city map. The simulation then advances in steps: at each step, the citizens assess how satisfied they are with their current neighborhood and may decide to move to a vacant address instead of remaining where they are. A citizen is satisfied in case a sufficiently large proportion of their neighbors belongs in the same demographic.

This model is based on the work of Thomas Schelling, a winner of the Nobel Prize in Economics. Naturally, it is a simplification of the real world; models are.

> Let me remind you of the particular characteristics of all of these behavior systems[.] It is that people are impinging on other people and adapting to other people. What people do affects what other people do.
>
> —Thomas Schelling

CitySim

The CitySim module contains two packages. The simulation model itself is in package `o1.city`, whose main contents are:

- `Simulator` (partial implementation given). You can ask a simulator object to launch a new simulation run (`startNew`) or advance the most recently launched simulation by one step (`moveResidents`). The simulator delegates some of its work to another object that represents the map of the city and is of the type:
- `CityMap` (ready as given). A city map is a grid, and `CityMap` inherits `Grid` much like `GameBoard` did in the Chapter 7.4's Viinaharava assignment. Each of the elements in this grid is of the type:
- `Demographic` (missing entirely). This simple sealed trait serves as a supertype for an `Occupied` class and a singleton `Vacant`, which represent occupied and vacant addresses, respectively.

The classes in `o1.city.gui` define the user interface, which you can safely ignore apart from the fact that `SimulatorApp` is an app object: it creates an instance of `Simulator` and calls its methods when the user presses the buttons in the GUI and slides the sliders.

Here's a diagram of these components:

../_images/module_citysim.png

Task description

Add the trait `Demographic` and its subordinate concepts `Occupied` and `Vacant`.

Add the methods `findDemographic`, `dissatisfiedResidents`, and `moveResidents` in class `Simulator`. (The `residents` method is also missing but we'll leave that for later.)

Run the program, experiment with different settings in the UI, and consider how they affect the results.

Instructions and hints

On `Demographic` trait:

- This is an exceedingly simple trait: it doesn't have any methods at all.
- `Occupied` and `Vacant` just need an `extends`, and that's about it.
- Seal the trait with `sealed` (Chapter 7.2). Once you do, any `Demographic` is guaranteed to be either an `Occupied` object or `Vacant`.
- The outcome is a set of types that resemble the `Option` class and its derivatives `Some` and `None`. And indeed we could have used `Option` here instead. However, `Demographic` and its subtypes communicate our model better.

On the `findDemographic` method:

- Note that the `cityMap` instance variable in `Simulator` gives you a reference to currently active map, which is a `CityMap` object that contains `Demographic` objects in a grid pattern.
- This method is pretty simple to implement once you find the right tools in the documentation of `Simulator` and/or `CityMap`.

On the `dissatisfiedResidents` method:

- A citizen's satisfaction depends on whether the percentage of similar citizens among the citizen's neighbors is high enough. The desired percentage, which the end user sets with a slider in the GUI, is available to you as a number between 0 and 100 in the variable `similarityDesired`.
- Every address does not have the same number of neighbors. Note the details in the Scaladocs.
- Try to split the method's overall task in subtasks. For instance, one subtask could be to examine whether the neighborhood of an address is unsatisfactory.
- Consider writing auxiliary functions for the subtasks. You can define them either as private methods or as local functions within `dissatisfiedResidents`.
- Again, make use of what the `CityMap` object gives you. Keep in mind that a `CityMap` is a sort of `Grid` and has all the methods it inherits from its superclass. For instance, there is a way to check which demographic currently occupies a particular `GridPos`.
- If you use division, remember that any and all decimals are discarded when you divide an `Int` by an `Int`.

On the `moveResidents` method:

- Make use of the two other methods you implemented.
- You may find `Random.shuffle` useful again (see Chapter 7.4), as well as other methods in class `Random` (see Chapter 3.6).

Additional hint for `moveResidents`

Show the hintHide the hint

> Here's an outline of a solution:
>
> 1. Form a buffer that contains the addresses of all vacant homes.
> 2. Form a collection that contains the locations of all the unsatisfied residents, in random order.
> 3. Repeat for each unsatisfied resident: Pick a random address from the buffer of vacant homes. Move the resident to that address on the `CityMap`. In the buffer, replace that destination address with the vacated address.

Use the app

Run the simulator on the default settings. Press Single Step repeatedly to advance the simulation. Try Run, too. Note that the satisfaction threshold is set at 70%, meaning that the citizens are very easily dissatisfied with their neighborhood.

Try higher and lower values for the threshold.

It seems obvious that if the citizens demand a great number of neighbors similar to themselves, they end up living among their own demographic. Something that's not equally obvious is how demanding the citizens need to be for the phenomenon to occur. Explore and find out.

Could we use a similar model to explain the "echo chambers" on social media?

What real-world factors are ignored by this model?

What would happen if one demographic cares about what their neighbors are like but the other is always or almost always satisfied? Or what if the citizens didn't just set a minimum but also a maximum for the degree of similarity between themselves and their neighbors?

Points C **60 / 60** My submissions **4 / 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 21 (CitySim 1)

Select your files for grading

Simulator.scala

Choose File No file chosen

Demographic.scala

Choose File No file chosen

Submit

In case this assignment piqued your interest

The book Networks, Crowds, and Markets: Reasoning About a Highly Connected World, which is available as a free online edition, will tell you more about computational modeling of social, economic, and medical phenomena, among other things. The Schelling model we just used features in Chapter Four of the book.

Reimplementing Auctions with Inheritance

The rest of this chapter consists of an assignment that revisits our earlier auction-themed programs. The programming assignment itself is optional, but we highly recommend that you at least read what it's about.

In Chapter 5.1, you presumably wrote `FixedPriceSale` and may have also written `DutchAuction` and `EnglishAuction`. These classes represents items put up for sale in a variety of ways. Then, in Chapter 5.5, we designed `AuctionHouse` to represent auction houses where all the items are sold in the traditional "English" style.

You can use your own earlier solutions as a basis for the upcoming assignment. If you didn't do some or all of them, feel free to use the example solutions (FixedPriceSale, DutchAuction, EnglishAuction).

A new class hierarchy

Here's how the existing classes relate to each other:

../_images/module_auctionhouse1.png
In other words: an `AuctionHouse` contains `EnglishAuction`s. The classes `FixedPriceSale` and `DutchAuction` are unconnected to the others.

In this assignment, you'll refactor the classes. The purpose of refactoring is to improve program quality: you'll modify `FixedPriceSale`-, `EnglishAuction`, and `DutchAuction` so that it's easier to use them all in combination. At the same time, you'll eliminate a great deal of redundant code. In this exercise, inheritance (Chapter 7.3) will be your main refactoring tool.

The goal is a hierarchy of classes that looks like this:

../_images/module_auctionhouse2.png
At the heart of our plan is the abstract class `ItemForSale`, which will serve as a generic superclass for items being sold in all sorts of ways. We'll be able to use this superclass to write a more generic `AuctionHouse` class. We'll also introduce an `InstantPurchase` class to capture what fixed-price items and Dutch-style auctions have in common.

Implement the refactoring

Implement `ItemForSale`, `EnglishAuction`, `InstantPurchase`, `FixedPriceSale`, `DutchAuction`, and `AuctionHouse` so that they match the documentation provided in module AuctionHouse2.

Instructions and hints:

- We recommend implementing the classes in the order listed above.
- As you read the Scaladocs, be sure to note which classes and methods are abstract and which methods each class inherits from its superclass(es).
- Just like in Chapter 7.3: if a superclass already defines a concrete instance variable, don't repeat the `val` in the subclass. For instance, the `description` variable is defined in the superclass `ItemForSale`, so don't redefine it as a `val` in the subclasses, even though the subclasses do need a description as a constructor parameter.
- You can use the given test app to try some of the key methods. You'll notice that the app object `o1.auctionhouse.gui.TestApp` generates a bunch of error messages to begin with, but they'll vanish once you make the requested changes.

  ../_images/auctionhouse2_gui.png
- In the `AuctionHouse` class, you'll need to replace `EnglishAuction` with a more general type, but that's the only change needed there.

Points training **0 / 0** My submissions **0** ▾ 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 22 (AuctionHouse Refactored)

Select your files for grading

AuctionHouse.scala

Choose File No file chosen

DutchAuction.scala

Choose File No file chosen

EnglishAuction.scala

Choose File No file chosen

FixedPriceSale.scala

Choose File No file chosen

InstantPurchase.scala

Choose File No file chosen

ItemForSale.scala

Choose File No file chosen

Submit

Once you finish the assignment, pause for a moment to admire the results: inheritance turned the disconnected and redundant classes into a beautiful conceptual model. The definition of each concept (class) includes only what is necessary to distinguish it from related concepts.

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 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 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 for this page

The assignment on Schelling's model of emergent social segregation has been adapted from a programming exercise by Frank McCown.

a drop of ink