Informatics 1: Object Oriented Programming

Assignment 3 –Modelling a zoo

The University of Edinburgh
David Symons (dsymons@exseed.ed.ac.uk)

Overview

This assignment aims to give you some practical experience of working on a larger project with multiple classes. The basic section focuses on modelling a scenario using inheritance to create a hierarchy of classes. The intermediate tasks revolve around data representation and the advanced section deals with a search problem.

Marks will be awarded out of 100 and count for 40% of your overall grade for Inf1B. Each 20 credit course is designed to involve around 200 hours of work. Of those, 20 hours have been allotted for this assignment. Completion time may vary depending on your level of experience and the grade you are aiming for. While you are encouraged to aim high, the intermediate and advanced sections are optional and grades in the upper bands are expected to be rare. Details on marking criteria can be found in the eponymous section below.

Before you start, please read carefully the section on *Good Scholarly Practice*. The final section gives instructions on how to submit your work, which is **due by 16:00 on Friday 16th April**. Please note that the Covid-19 extension has already been applied and **no further extension is possible!**

Basic: Building a zoo

The first task is to model a zoo based on the following description.

A zoo consists of multiple areas, each of which can be identified by a unique ID. There must be exactly one entrance area, which always has an ID of zero. Any number of picnic areas and animal habitats may be added to (or removed from) the zoo. Aquariums can accommodate seals, sharks and starfish, while lions, gazelles and zebra are kept in enclosures. Buzzards and parrots must be caged.

Each habitat is assigned a maximum capacity upon creation. This maximum number of animals may not be exceeded when populating the zoo. Attention must also be paid not to put incompatible animals into the same habitat. Zebra and gazelles can be in the same area, but neither can live with a lion. Sharks cannot be allowed near the seals, but starfish are compatible with both. The birds do not get along.

Your aim is to create and implement the required classes. The use of inheritance and a well chosen class hierarchy is essential! If done correctly, this will significantly reduce code duplication. The steps below guide you towards a clean design.

Step 1: Download the provided code

Starter code is provided in the form of *src.zip*, which can be downloaded from LEARN (under Assessment → Assignment 3→ src.zip). Create a new Java project in your favourite IDE and replace the *src* folder with the one provided.

Step 2: Create the animal classes
Create a class for each animal *(Lion.java*, *Zebra.java*, etc.) inside the *animals* package. All of these must be related to the provided, abstract class *Animal.java*. As you can see from this class, every animal has a *getNickname()* method, which returns the name given to the animal upon creation. Think about where it is best to implement this method and where the name should be stored. Now, implement the *isCompatibleWith()* method for each animal according to the above restrictions on sharing a habitat. The method need not cater for animals that cannot live together anyway.

Step 3: Create the different areas
Create: *Entrance.java*, *PicnicArea.java*, *Aquarium.java*, *Cage.java* and *Enclosure.java* inside the *areas* package. These classes must all conform to the *IArea* interface, but you may find it useful to create some intermediate classes in the hierarchy. For now, *getAdjacentAreas()* can return null.

Step 4: Create the zoo
Create *Zoo.java* in the *zoo* package and have it implement the provided interface *IZoo*. This will require adding all unimplemented methods to your code. The ones that are not needed (yet) should return null, zero or false. This is to ensure your code will compile.

Step 5: Add new areas to the zoo
Implement the *addArea*, *removeArea* and *getAre*thods in *Zoo.java*. To do so, you will need to decide on how to store and retrieve the areas that are added. Depending on your design you may (or may not) need to change other classes. You must also choose some way of generating unique integer IDs for each area. Finally, make sure the zoo always has an entrance area with ID zero.

Step 6: Add animals
Now implement the *addAnimal* method. This method returns a byte code that indicates if the operation was successful or why it failed. Error codes are provided in *Codes.java*. When multiple reasons apply, the error with lowest value should be returned (as documented in *Codes.java*). The *addAnimal* method should take advantage of your other classes by delegating some of the responsibility. You may find that some changes to your current design and class hierarchy are required to do so.

Step 7: Report on your design decisions
Explain your design with a particular focus on where, how and why you made use of inheritance. Include this in the *Basic* section of your report. Guideline length is 300-500 words.

Note on reporting issues
Should you have difficulties completing any of the coding steps, you can explain your ideas in your report to get some of the credit. Please do so under separate sub-heading e.g. “Issues with step 5”. You can also report problems with partially working code, such as when it fails only for certain inputs. These sections are not required if your code is fully functional. Since the problems can vary greatly, a fixed word limit does not make sense here, but you should try to be as concise as possible.

Intermediate: All around the zoo
Due to an ongoing pandemic, the zoo has to adopt a one-way system for visitors moving between areas. Currently all paths have been blocked off and it is your job to reconnect them and allow visitors to go between the different areas. The following methods need to be implemented.

connectAreas(int fromAreaId, int toAreaId)
After this method is called, visitors are allowed to go from the area with the *fromAreaId* to the area with the *toAreaId*. Going the other way is not possible (unless you call the method again with the

arguments swapped). As part of this, you now need to implement the *getAdjacentAreas()* method of *IArea.java*. When an area is deleted, any connections from or to it should also be removed.

isPathAllowed(ArrayList<Integer> areaIds)
Returns true if (and only if) visitors are allowed to visit the areas in the order given by the passed in list. The path starts with the area ID at index 0 in the list.

visit(ArrayList<Integer> areaIdsVisited)
A visitor passes through the rooms in the order they are listed in *areasIdsVisited*. They write down a list of all animals they encounter in the order they see them. Within each area, the visitor always happens to spot the animals in the order they were added to the habitat. The list of names is returned by the method. The visitor may start taking notes from any area in the zoo (need not start at the entrance). However, if they disobey the one-way system while taking notes, the notes are confiscated and null is returned.

findUnreachableAreas()
This method returns an ArrayList of all area IDs belonging to areas that cannot be visited when starting from the entrance area. The listed area IDs can be in any order.

Report
The intermediate section of your report should cover the following under separate sub-headings.
1. Explain how you modelled the zoo's areas and connections. (Guideline 300 words)
2. Describe an alternative representation and explain why you did not choose it. (Guideline 200 words)

Advanced: Ticket machine

The zoo offers ticket machines at which visitors can pay the entrance fee in cash. The machines need to be programmed to return the correct change. Your task is to implement the *payEntranceFee* method of *IZoo.java* along with a few other (mostly trivial) methods described here:

setEntranceFee(int pounds, int pence)
As the name suggests, this is used to set the price of a ticket in pounds and pence. Note that you should not use floats or doubles when dealing with currency. This is due to precision issues with those types. After some numeric operations, a result that should e.g. be exactly £1 can be 0.9999.

setCashSupply(ICashCount coins)
This method is used to stock the ticket machine with cash that it can return as change. This will require implementing the *ICashCount* interface, which is used to represent an amount of cash in terms of the numbers of the following notes and coins: £20, £10, £5, £2, £1, 50p, 20p and 10p. The machine does not accept 1p or 2p coins or notes larger than £20.

getCashSupply()
Simply returns the *ICashCount* instance, which should reflect the number of notes and coins currently in the ticket machine. These counts may increase as a result of cash being inserted or decrease when change is returned.

payEntranceFee(ICashCount cashInserted)
The challenging part is this method, which takes as its parameter the cash that is inserted and returns another ICashCount instance representing the correct change. The contract of the method is as follows:

- If the inserted cash amounts to less than the price of a ticket (as set via *setEntranceFee*), the machine returns the cash that was inserted (not just any equivalent amount of cash, but the exact same number of each note and coin).
- If too much money is inserted and the machine cannot return the exact change (due to running out of the required notes or coins), the machine returns the cash that was inserted (again the same number of each note or coin).
- If the inserted cash pays for the ticket exactly, no change is given. This is indicated by returning an *ICashCount* instance that contains zero of each denomination. The inserted cash is added to the machine's cash supply.
- If too much money is inserted and the machine is able to give exact change, it must do so. So as not to annoy the visitor by paying out the entire change in 10p coins, the machine will always prioritise large denominations to return the least number of notes and coins possible (given its current supply). The cash inserted by the user can be used to make up the change.
- Example: A user pays for a £17.80 ticket with a £20 note. Coins adding up to £2.20 must be returned as change if at all possible. If the machine has a £2 coin and a 20 pence piece, then those must be given. If, however, the machine is out of 20p coins it returns two 10p coins. If it does not have those either, it returns the £20 note (no sale).

Testing
You will have to write your own tests for checking the correctness of your implementation as adhering exactly to the specification is part of the challenge. Your tests will not be assessed, but the correctness of your algorithm will be.

Report
The advanced section of your report should cover the following under separate sub-headings.
1. Explain how you chose to represent money/prices and why. (Guideline 100 words)
2. Explain the key ideas behind your algorithm. (Guideline 300-500 words)

As always, you can (and should) use the report to document any attempts you made. This is especially important with automated testing, as the tests will either pass or fail and cannot detect code that is almost correct. Even if you only had some ideas for how to approach the task, they are worth writing down.

Restrictions
The following restrictions apply to this assignment:
- Do not use any functional language constructs such as lambdas or streams.
- Do not use any third party libraries.

Good Scholarly Practice
As with all assessed work, you must fulfil the University's requirements for good scholarly practice. Key rules include:
- Any work that is not your own must be clearly identified and referenced. The only exception is the material provided as part of the assignment.
- Any code that is not your own must be clearly identified and referenced. Also, you will not receive any credit for code that anyone can look up! If you have significantly modified the code you can get some credit, but you still need to reference it and explain exactly what your contribution is. This is seldom worth the effort.
- Any text not written by you must be quoted and referenced.
- Any figures, images or graphics not created by you must be referenced.

The full misconduct policy on https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct takes precedence over this incomplete summary which is only provided for your convenience.

Marking Criteria

Marks will be assigned in accordance with the University's Common Marking Scheme (CMS).
See: https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme

For a pass grade (up to 49%) it is sufficient to tackle the *Basic* section. This includes both code and report aspects, with the latter playing a significant role. Code quality is also taken into account, but to a lesser degree than in higher grade bands. The same holds true for the presentation of the report.

For a 2nd or low 1st class (up to 79%) you must first complete the the basic report and almost all basic coding features to a high standard. Only then does it make sense to address the *Intermediate* tasks (code and report). Quality and presentation become more important. There should be evidence of a conscious effort to produce good code and a clear, informative and structured report.

The advanced section is for those aiming at an A1 or A2 (up to 100%). At this point, all basic tasks must be well polished and almost all intermediate tasks should well solved and documented. Grades in this range require excellent code quality and a clear, well presented report that demonstrates real insight.

Code quality

The assessment of code quality includes the following aspects:

- Structure
  - A good design should split the task into more manageable chunks (think divide and conquer) e.g. by a meaningful division of the code into classes.
  - Modularity and reusability: Classes and methods perform well defined tasks. They can be seen as building blocks for solving many different problems, not just the task at hand. Long methods are a symptom of a too problem-specific design.
  - Code duplication is avoided.
  - Extensibility: New features can be added without redesigning the whole program.
- Robustness
  - The program is able to handle errors and invalid/unexpected inputs without crashing.
  - Correct use of access level modifiers (public/private) and static to control exposure.
- Technical/language proficiency
  - Use the appropriate features of your programming language. This means not restricting yourself to building everything using a few keywords when there is a more elegant solution. It also means not using a needlessly complex mechanism for solving a simple task.
- Readability (self-documenting code)
  - Use descriptive but concise names for variables, methods and classes.
  - Class and interface names are in PascalCase (first letter uppercase).
  - Method and variable names are in camelCase (first letter lowercase).
    - Exception: Static constants are written in SHOUT_CASE.
  - Use consistent formatting throughout all classes.
- Commenting
  - Quantity: Comments should be used where they are useful and nowhere else. Comment as much as you feel is needed for someone else to be able to understand your code – or to make sure you yourself can understand it when you come back to it a year later. Do not clutter really simple bits of code (that are self-documenting) with needless comments.
  - Quality: Comments should provide additional insight into what is happening in the code. They should not just re-state what the code is doing.

Report

The assessment of the report includes the following aspects:

- Content
  - Answer questions directly and to the point.
  - Do not write down everything you know about the topic, stick to what was asked.
  - Go beyond giving the correct answer by explaining why it is correct (shows understanding).
- Structure
  - Break the content down into well labelled sections.
  - If you are making an argument, ensure you organise your points into a logical sequence.
  - Try to make the text flow by avoiding frequent or abrupt changes of topic.
- Writing style
  - Should be clear and concise.
  - Make the report as short as possible without sacrificing content.
  - Avoid repetition: Make your point once and make it well, but avoid re-stating it.

Please note that markers are not required to read much beyond the guideline lengths for each report section. This will result in you not receiving credit for all the ideas you expressed, so it is in your interest to keep it brief.

Submission

This assignment is **due at 16:00 on Friday 16th April.**

For this assignment, the Covid-19 extension has already been applied. No further extension is possible and late submission will result in a zero grade!

Your code and report are submitted to CodeGrade, a tool that automatically tests your code for functional correctness and robustness. It is accessed via LEARN (under Assessment → Assignment 3 → CodeGrade). Once the interface opens, you can click on "New tab" (on the bottom left) to enlarge the view.

Please practice submitting well before the deadline. New submissions will overwrite previous ones and you can submit as often as you like. You will receive automated feedback on your progress and can adapt your solution to achieve a higher score.

Keep in mind that the auto grader only tests for functional correctness and robustness. While these scores contribute to your grade, your marker will also consider your report and all other aspects listed in the *Marking Criteria* section.

For automated testing to work, your code must compile. This is why it is essential that you provide dummy implementations for all methods, even if you do not intend to solve the corresponding tasks. All you have to do is return null, zero or false in the method body.