SYSC2004 Fall 2018

Assignment 3

Objectives:

- To practice writing and subclassing abstract classes
- To practice the use of `HashMap` as a database
- To practice reading and writing files

Posted: EFdictionary.csv

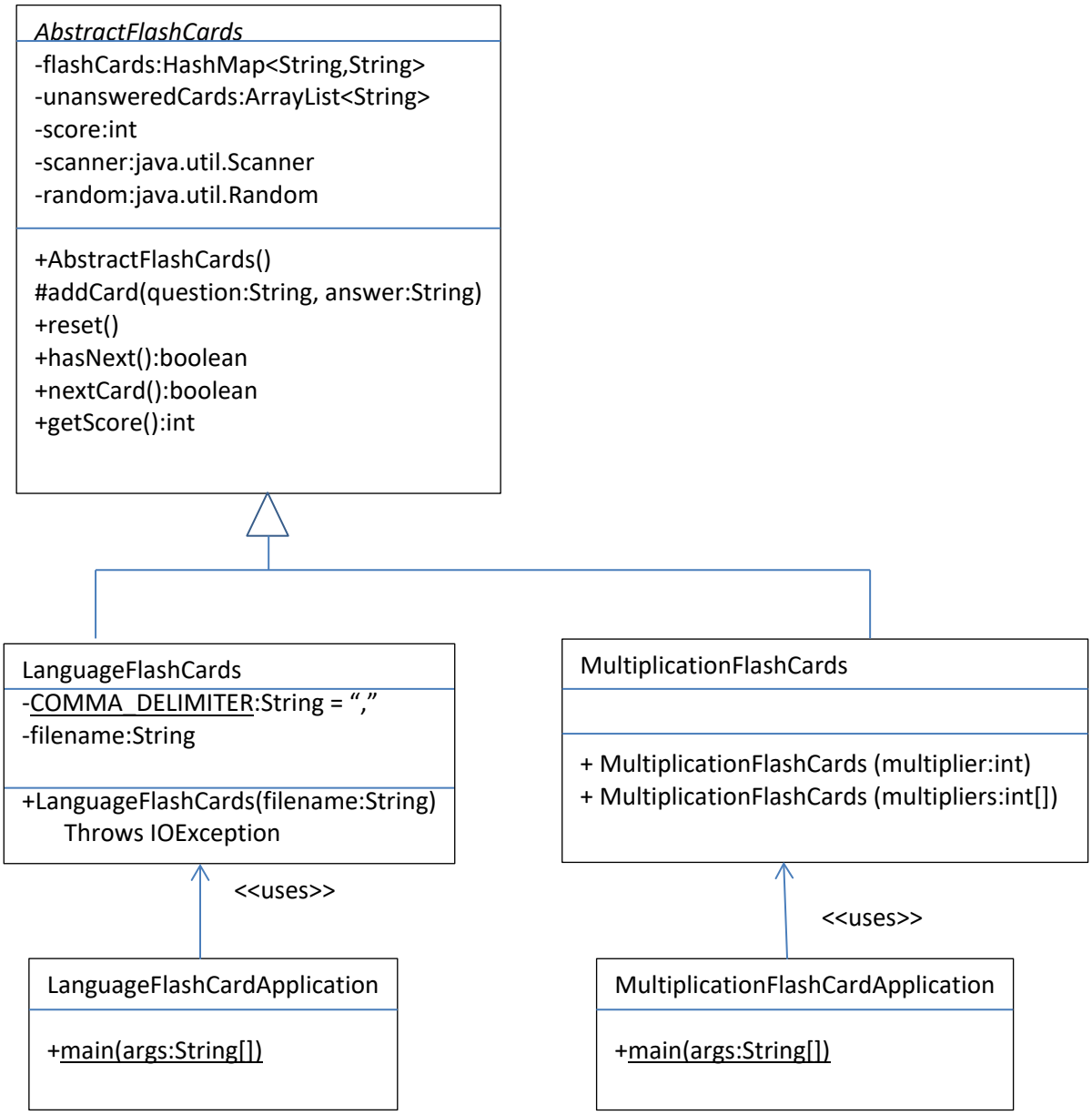
SUBMISSION: AbstractFlashCards.java, MultiplicationFlashCards.java, LanguageFlashCards.java, MultiplicationFlashCardApplication.java and LanguageFlashCardApplication.java

1. Rules of the Game

Flashcards are used to help some memorize a set of facts. They offer students repetitive practice in a particular set of facts. Each flashcard has a question on its front (Example: 4 + 8 = ?) and the correct answer on its back (Example: 12). The first flashcard is revealed and the student attempts to answer correctly. If the answer is correct, this particular flashcard is set aside (so that the student is not asked something that they already know again) and the student gains a point in their total score. Instead, if the answer is incorrect, this flashcard is re-inserted, randomly, back into the deck of flashcards, and no point is earned. Play continues until all flashcards are correctly answered, or until the student tires and quits.

Flashcards are used in many subject matters – to practice math facts (e.g. addition and multiplication tables), to practice language vocabulary (e.g. from French to English, or vice versa), to master all the bones in the body in anatomy class. The software that we write should be extensible to allow the implementation of this variety of flashcards. An abstract class hierarchy will be used to demonstrate one possible design approach (and to give you practice in using abstract classes).

The UML Class Diagram.



Explanation of the UML class diagram

1. `AbstractFlashCards` is a parent class incorporating all that is common between different varieties of flashcards, whether they are used for math, language or anatomy. The class is abstract even though all of its methods are concrete. The rationale is that even though it is known how to implement the methods, we don't know fully how to create the deck (i.e. we need more constructors) so therefore we don't want clients to instantiate this class.
    - Look at its concrete subclass called `LanguageFlashCards`. Its constructor has an argument called `filename`, suggesting that these cards are to be read in from a file.
        - The file (EFdictionary.csv) to be read is posted as part of the assignment. Look at its contents. It is a simple csv file. Writing this class will provide you with some more practice reading files.
    - Look at its other concrete subclass called `MultiplicationFlashCards`. Its constructor(s) has(have) an argument called `multiplier`(s), suggesting that these cards will not be read in from a file, but will be automatically generated for a given `multiplier` (Example: If the single multiplier is 10, twelve cards will be automatically generated for 1x10, 2x10 … 12x10. If instead an `array of multipliers` is provided (say, [2,5,7]), thirty-six cards will be generated: 12 for 2, then 12 for 5, then 12 for 7
        - In the UML class diagram, take note that this subclass has no instance variables. You will simply write programming logic to automatically generate the described sequences of cards.
2. The instance variables of `AbstractFlashCards`
    - `flashcards` is a hashmap (i.e. a dictionary) with the `question` on the front of the flashcard as the key and the `answer` on the back of the flashcard as the value. It contains the original complete deck of flashcards.
    - `unansweredCards` is a randomly-ordered copy of all questions (because you can look up the `answer` later when the `question` is used.). A separate randomized copy is needed because the `flashcards` stored in the hashmap have a specific ordering; simply iterating over the hashmap will lead to a very boring game. Originally, it will contain the complete deck in random order. The first element (index = 0) is the next `question` to be presented. If the student answers it correctly, the `question` is removed from this list; if the student answers it incorrectly, the `question` is re-inserted at a random location within this list. The game is over when this list is empty.
    - `score` is a simple integer number incremented whenever a correct answer is given.
    - `scanner` is necessary for the class to read in answers from the console.
3. The methods of `AbstractFlashCards` are all concrete
    - The constructor shall initialize all instance variables. These variables are initialized even though you can't actually fill the decks with concrete cards.
    - `addCard`() is a protected method. Use of `protected` is a signal that this method is intended for use by any subclass. Its role is to put a <question,answer> entry into the

`flashcards` hashmap. (Sample Exam Question: Why can't the subclasses do it directly? Why is this protected method needed?)

- `reset`() does not alter the existing `flashcards`; instead it is used to re-shuffle the entire deck, meaning that it clears out the `unansweredCards` and creates a new random ordering of the `questions`.
    - This is a programming challenge for you to conquer, to push the limits of your problem-solving skills. If you cannot figure it out, post a question on this Assignment's CULearn discussion forum.
    - Handy syntax: To convert any Java Collection (e.g. aCollection<String>) into an equivalent array.

      ```
      String array[] = aCollection.toArray(new String[aCollection.size()]);
      ```

    - Handy syntax: To construct an `ArrayList` that is a copy of an existing array, there is the one-argument constructor (instead of the usual default constructor which creates an empty list)

      ```
      arrayList = new ArrayList(array);
      ```

- `hasNext`() returns true if there are any remaining `unansweredCards`.
- `nextCard`() is the heart of the game. The method prints the next `unansweredCards` and reads in the user's answer. If the answer is correct, the card is removed from the list of `unansweredCards`, the user's `score` is advanced, and the method returns true. If the answer is incorrect, the card is re-inserted – at a random place – back into the list of `unansweredCards`, and the method also returns true. The method only returns false if there are no remaining `unansweredCards`.

Writing the Code

Because of the abstract class (which you cannot instantiate to test), you will have to write more than one class at a time. When presented with a large task, it is always good to work incrementally. Don't write the whole class; write parts of the class and then test them, before proceeding to write more.

The two games are very similar. Complete one game (e.g. the LanguageFlashCardsApplication) so that you understand the game fully. Then write the other one, because you will only have to write code for the differences.

Tip: In this assignment's project, you will have two classes with a main() method. Instead of using the RUN menu (which launches the main() method of the alphabetically-first class in the project), you will have to identify WHICH main() to run.

- On the right-side PROJECT window, right-click on the desired xxxApplication.java and select RUN.

On the next two pages are the expected console dialogues that you will have to replicate with your programs, one for `LanguageFlashCardsApplication` and one for `MultiplicationFlashCardsApplication`. The two are very similar, except for the content of the cards and for the initial prompts about how to create the cards.

LanguageFlashCardApplication : Sample Console Input/Output that you are to replicate

- Blue are the console output to be printed in main() to direct the sequence of the game.
- Black are the console inputs entered by the user (i.e. you)
- Red are console outputs used for supporting testing, to be printed by the AbstractFlashCard::nextCard() method. It is used to prove that you are presenting the cards in random order, and that after a wrong answer, you are inserting the card back into the pile at a random location.

What is the filename containing your flashcards? Exact letters!
EFdictionary.csv
[country, chair, book, one, horse, school, car, cake, cat, ten, dog, sugar, map]
country
pays
You're correct!
Score = 1
Next? (Y or N)
y
chair
chase
Sorry, please try again
[book, one, horse, school, chair, car, cake, cat, ten, dog, sugar, map]
Score = 1
Next? (Y or N)
y
book
livre
You're correct!
Score = 2
Next? (Y or N)
n
You've got a score of 2 so far

MultiplicationFlashCardApplication : Sample Console Input/Output that you are to replicate

- Blue are the console output to be printed by main(), to direct the sequence of the game.
- Black are the console inputs entered by the user (i.e. you)
- Red are console outputs used for supporting testing, to be printed by the AbstractFlashCard::nextCard() method. It is used to prove that you are presenting the cards in random order, and that after a wrong answer, you are inserting the card back into the pile at a random location.

Which times tables would you like to test? (Between 1 and 12 inclusive)
3 6 7 9
[6 * 10, 3 * 8, 3 * 7, 3 * 9, 3 * 4, 3 * 3, 3 * 6, 3 * 5, 7 * 12, 3 * 2, 3 * 1, 7 * 11, 7 * 10, 7 * 5, 7 * 6, 7 * 3, 7 * 4, 7 * 1, 9 * 12, 7 * 2, 9 * 10, 9 * 11, 7 * 9, 7 * 7, 7 * 8, 6 * 8, 6 * 9, 6 * 1, 6 * 2, 6 * 3, 6 * 4, 6 * 5, 6 * 6, 6 * 7, 9 * 3, 9 * 4, 9 * 1, 9 * 2, 9 * 7, 9 * 8, 9 * 5, 9 * 6, 9 * 9, 3 * 12, 6 * 11, 6 * 12, 3 * 11, 3 * 10]
6 * 10
60
You're correct!
Score = 1
Next? (Y or N)
y
3 * 8
35
Sorry, please try again
[3 * 7, 3 * 9, 3 * 4, 3 * 3, 3 * 6, 3 * 5, 7 * 12, 3 * 2, 3 * 1, 7 * 11, 7 * 10, 7 * 5, 7 * 6, 7 * 3, 7 * 4, 7 * 1, 9 * 12, 7 * 2, 9 * 10, 9 * 11, 7 * 9, 7 * 7, 7 * 8, 6 * 8, **3 * 8**, 6 * 9, 6 * 1, 6 * 2, 6 * 3, 6 * 4, 6 * 5, 6 * 6, 6 * 7, 9 * 3, 9 * 4, 9 * 1, 9 * 2, 9 * 7, 9 * 8, 9 * 5, 9 * 6, 9 * 9, 3 * 12, 6 * 11, 6 * 12, 3 * 11, 3 * 10]
Score = 1
Next? (Y or N)
y
3 * 7
21
You're correct!
Score = 2
Next? (Y or N)
n
You've got a score of 2 so far

Marking Scheme (Total : 64 – 16*4)

Assignments will not be accepted if standard indentation is not used. A mark of zero will be earned.

Instruction to TAs: Make sure your write your initials on each assignment that you mark, so that the student knows who to contact for concerns about their mark.

Instructions to Students: Afterwards, if you have a question about the marking of your assignment -

1. On the course webpage, study the schedule for the TAs and see when your TA is on duty in the lab. If you do not have a class/lab conflict with that time, visit the TA during their lab period.
2. Alternatively, email the TA that marked your assignment. Pose your question if it is simple; if not, request a meeting. If you do not receive a reply from the TA within 2 working days, re-send your email and now 'cc your instructor so that s/he can follow up.

Marking Values: Each criterion is worth 4 marks and will be marked according to the rubric levels of BDAE (0 is not present at all, or consisting only of the UML call signatures)

Beginning for 1 mark: A bit but mostly not.

Developing for 2 marks: Some but missing key points and/or ignoring requirements & conventions

- Maximum mark possible if the code does not work.

Accomplished for 3 marks: Mostly of the vital learning points demonstrate mastery

Exemplary for 4 marks: Professional level code that is even better than asked.

AbstractFlashCards

/4 Class is an abstract class with concrete methods with all call signatures corresponding to UML (protected) Constructor properly initializes all instance variables

/4 reset() creates a randomly ordered copy of the questions, and resets all relevant instance variables

/4 nextCard() operates as described including re-insertion in random order of incorrectly answered questions.

LanguageFlashCards

/4 Correct construction of cards by reading in from a file

MultiplcationFlashCards

/4 Correct construction of cards for 12x tables, with two constructors.

Both Application classes

/4 main() method is written as described, matching the provided scripts.

Overall Code Inspection:

DOES the code adhere to the Java Style Guidelines? TBD (https://learn.zybooks.com/zybook/CARLETONSYSC2004SchrammWinter2018/chapter/2/section/8) . /4 - For Whitespace – Appropriate, consistent whitespace between lines of methods as well as spaces between operators and arguments (Example: x = 5; instead of x=5;). Readability.

/4 - For Braces – consistent placement of { } for readability

/4 - Naming of any working variables using conventions

/4 – Efficient use of variables – avoidance of repetitive allocation of variables through judicious use of variable scope.

/4 * 2 – Well-constructed JavaDoc for all methods in all classes.