COMP1010 Fundamentals of Computer Science

Assignment 4

Last updated: May 23, 2021

Assignment Details

- The assignment will be marked out of 100 and it is worth 5% of your total marks in COMP1010.
- Due date: Sunday, 6 June 2021 at 21:00 AEST.
- Late penalty: 20% per day or part of (if you submit an hour late, you will be given a 20% penalty).
- Topics assessed: Linked List, Classes and Objects

Background

For your final assignment, you are going to write an implementation of a word-solitaire game using linked lists, and since many of you probably have not seen this kind of game before, I will start by explaining how the game works.

The objective of the game is to gain score by creating words from letter tiles, just like in a game of Scrabble. The game board consists of four main components as depicted in Figure 1: the *deck* button, the *hand*, the *word*[1] and the *score* button. At the start of the game, both the player's hand and word are empty, as shown in Figure 2.

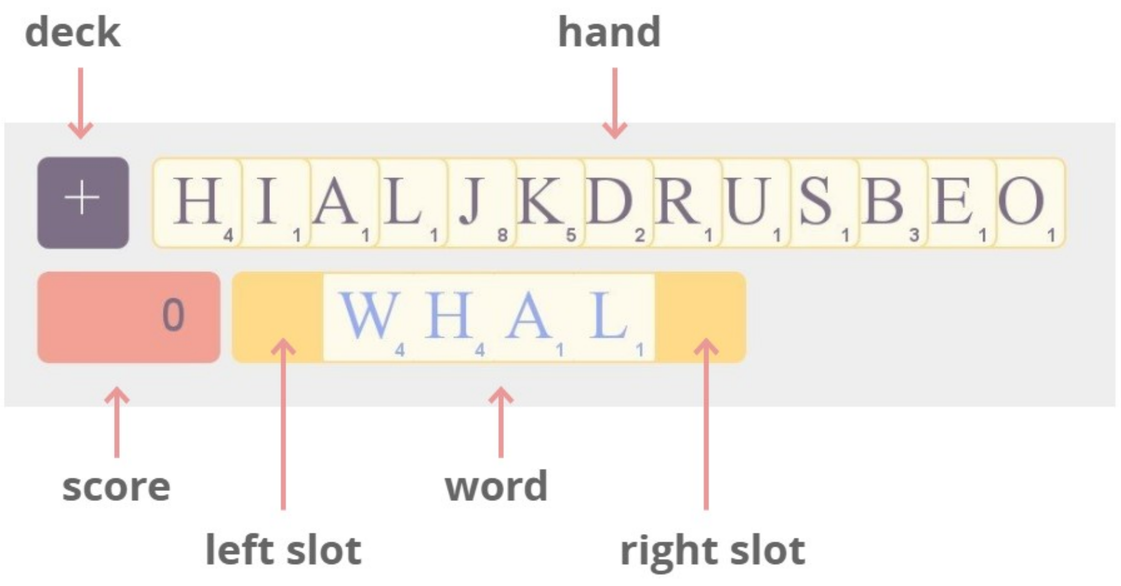

Figure 1: The game board.

The player obtain letter tiles by clicking on the deck button, which will add a letter tile to the player's hand (we refer to this as drawing a tile from the deck). Letter tiles are added to the left of the other tiles in the hand (see Figure 3). Player use the letter tiles to form a word, but the main rule of the game is that the player can only move the leftmost letter tile from the hand, and can only extend a word by adding the letter to the start or to the end of the word. Letter tiles already added to the word cannot be removed.

[1] Even though we call the component *the word*, it may not actually be a valid word because the player construct it one letter at a time. For the rest of the document, I will try to differentiate between the two by using the term *valid word* to refer to an actual word, while using *word* to refer to the player's word (which may not be a valid word).

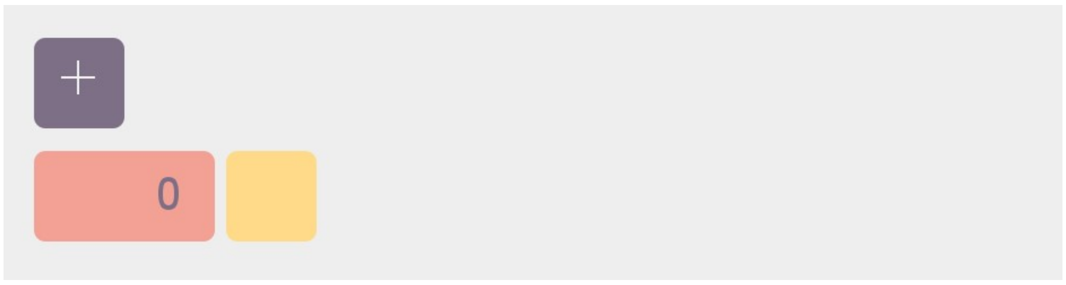

Figure 2: Board state at start of the game

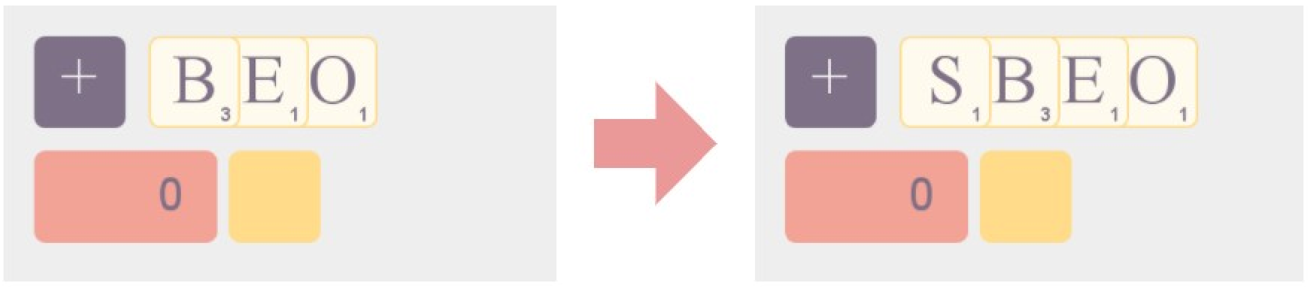

Figure 3: Drawing a tile from the deck. Tiles are added to the left of existing tiles in the hand.

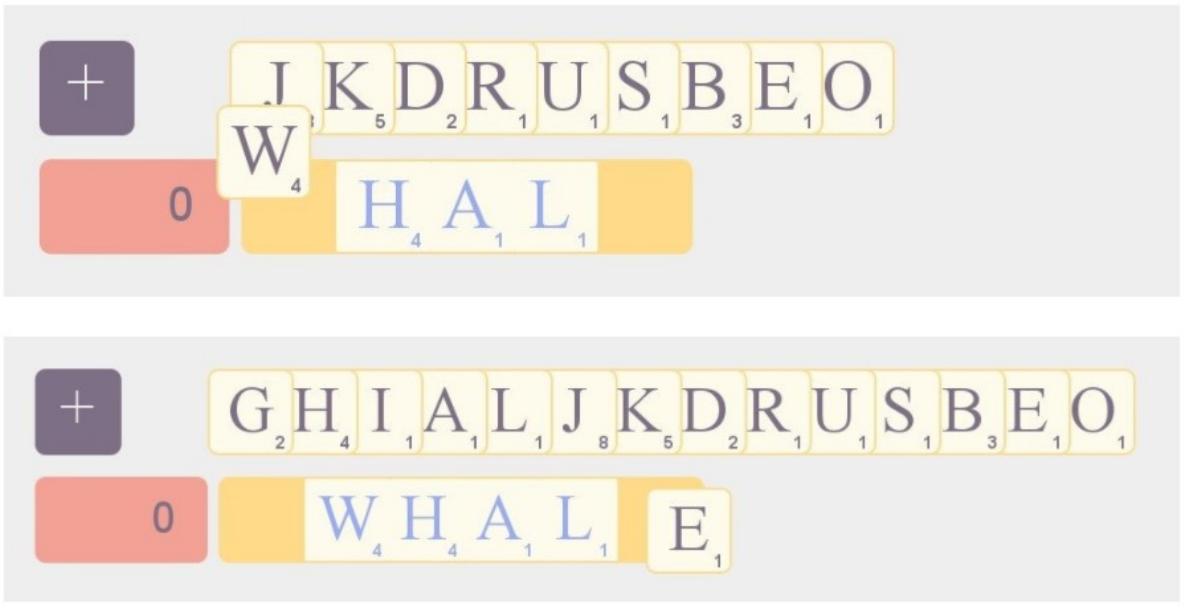
+
J K5 D2 R1 U1 S1 B3 E1 O1
W4
0
H4 A1 L1
+
G2 H4 I1 A1 L1 J8 K5 D2 R1 U1 S1 B3 E1 O1
0
W4 H4 A1 L1
E1

Figure 4: Tiles can only be added to the start or end of the word

Of course when the word is empty there is only one place to put the leftmost letter (the yellow box), but once the player have moved at least one letter to the word, then the player can add more letters by moving the letter to either the left or the right slot (corresponding to the start and end of the word respectively).

Once the player has formed a word with at least three letters, the player can click on the score button to *score* the word, that is, if the word is a valid word, then it will be removed and its score added to the score total. The score of a word is obtained by adding the score on each tile (see Figure 5).

However, if the word is not a valid word, then there are two possibilities. If the word can be extended to a valid word, then no point will be added to the score, but the word will remain on the board. For example, if the player tries to score the word WHAL, then nothing will happen, because we can extend WHAL to WHALE (among other words). On the other hand, if the word cannot be extended to form a valid word, for example, WHL, then the word will be discarded and no points will be awarded. A player can choose to discard the current word by adding letters so that it becomes invalid and then clicking on the score button.

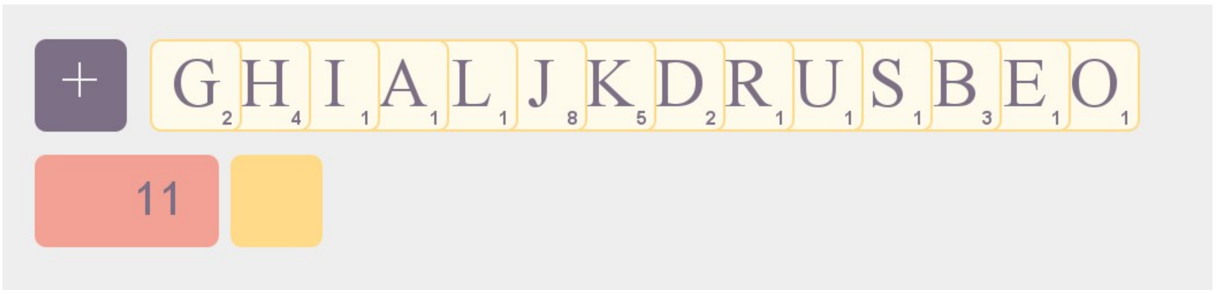

Figure 5: Clicking on the score tile scores the word, adding points to the total score, and removes the word. For example, the word WHALE adds up to 11 points.

The game ends when there are no more letter tiles in the hand or the deck, although with the way I implemented right now, the game does not actually end. The player simply won't have any more moves to make and so must restart the game.

Your Task

Your task is to implement the logic of the game by completing the methods in the following four classes:

- `Deck.java`
- `Hand.java`
- `Word.java`
- `Game.java`

The first three classes correspond to the game components described earlier, namely the deck, the hand, and the word. The final class, `Game.java` controls the interactions between the other classes. The deck, the hand, and the word should all be implemented as a linked list of Letter objects, which is a representation of the letter tiles. The code for the Letter class, in `Letter.java`, is already written for you, and it is not part of your submission (so please do not modify it).

The linked list implementations of the deck, the hand, and the word have slight differences. The deck is implemented as a queue, that is, the first letter added to deck should be placed at the top of the deck (i.e. the first letter to be drawn). Conversely, the hand is implemented as a stack, since the first letter added to the hand will be the last letter that can be moved (the last letter added to the hand is placed leftmost, and so it is the only letter that can be moved by the player). We cannot remove any tile from the word, but we can add a letter to either end, so this is yet another kind of linked list that must be implemented differently.

To implement the linked lists, you are free to add more attributes (class variables) and methods to the four files that you have to submit, except that you are not allowed to use arrays unless the method requires it. You are also not allowed to import any packages. See the section Limitations further below for more details.

You can find the exact marks allocated for each method that you have to implement in Marks and Grade Distribution section further down in this document. The details of what each of these methods do can be found in the code template given to you. However, here is a general outline of the tasks that you need to complete and a rough guide on the order that you should attempt them:

1. Implement the add methods in for the `Deck`, `Hand`, and `Word` classes (`addToStart` and `addToEnd`), for a total

of 16 marks. Completing these methods allows you to start populating the linked list which makes it easier to test the other methods.

2. Implement the `size` methods for the `Deck`, `Hand`, and `Word` classes, for a total of 12 marks, which gets you to start traversing the linked list.

3. Implement the constructors for `Deck` and `Game`, for a total of 7 marks.

4. Implement `getScore`, `isWord`, and `isPossibleWord` in `Word.java`, and then `scoreWord` in `Game.java`, for a total of 16 marks.

Completing all the above methods should give you all the pass marks of 51 out of 80 (the marks distribution is shown later in this document). You may not be able to follow this exact order because there are inter-dependencies between the classes, so you are definitely free to complete the tasks in the order of your choosing. Once you complete these pass-level methods, you can start concentrating on the rest of the Credit and Distinction level methods.

Important: when you add a Letter to either the deck, the hand, or the word, or move a Letter between them, you should NOT make a new instance of the Letter. Instead you should just add the reference to the linked list. The only exception is the `add` method in `Deck.java` since you are not given a `Letter` as an input parameter, but the actual character and its score.

For the High Distiction level, one of the tasks require you to implement the method `toArray` in `Word.java`, but the other two tasks do not actually ask you to write any new methods. They only test if you have implemented the `size` and `add` methods efficiently, so you may not have to do any extra work!

Limitations

The assignment assesses your understanding of linked lists, therefore you are NOT allowed to use an array in any of your methods unless it is given as an input, or if you are required to return an array. The exceptions for the above rule are:

- The constructor for the Game class (since both input parameters are arrays).
- The constructor for the Deck class (since the input parameter is an array).
- The methods `isWord`, `isPossibleWord`, and `toArray` in class Word.

You may be severely penalised for using arrays instead of linked lists in any other methods. You must also not use the ArrayList class or any other Java container classes. In fact, *you are not allowed to have any import statements* in any of the classes that you are submitting for this assignment.

JUnit Tests

As in previous assignments, you are given a JUnit test (`UnitTest.java`) which you should use to ensure that your code meets all the requirements. The assignment will be automarked using a similar test suite. Obviously the values used in the tests will be modified when your assignment is being marked, but the tests themselves should be very similar.

Visualisation

For this assignment, I have added a visualiser that you can use to check if you have implemented the game logic correctly. As in the previous assignment, the graphics are implemented using the Java Swing library, and all relevant classes can be found in the `graphics` package. To run the game, you need to run the `GameBoard` class. Of course you need to finish some parts of the assignment before the visualiser starts working, and it will properly work only when you have finished every method.

The GameBoard class creates an instance of Game and uses it to access the game components (the deck, the hand, and the word). The visualisation is done by simply drawing whatever is in the game components after every move, so whatever letters you have in the deck, the hand, or the word, will be drawn on screen. The implementation is much more complex than the one I did for the previous assignment, so I will not go into the details. However, here is something that may help you complete the assignment; the code that is used to create the dictionary and the array of letters for the game in the following lines inside the main method in GameBoard.java:

```
public void run() {
  ...
  String[] dictionary = Dictionary.standardDictionary();
  Letter[] letters = LetterGenerator.standardLetters(9999);
  // Letter[] letters = LetterGenerator.small(9999);

  GameBoard b = new GameBoard(dictionary,letters);
  ...
}
```

The dictionary is based on the file ukenglish.txt that comes with the assignment and you can see how it is constructed in Dictionary.java. I do not think that you have any reason to modify the dictionary (unless you do not like the dictionary and want to use your own), so I suggest leaving it as it is. On the other hand, the standard array of letters contain 100 letters, so if you would like to use something smaller to debug your code, you can use a smaller one (LetterGenerator.small(9999)) with only 12 letters. The integer parameter is just the seed for the random number generator used in randomising the letters. You can see how the array of letters is constructed in LetterGenerator.java.

The rest of this section is an optional reading, because you do not need to understand how the visualiser works to complete the assignment. However, if you want to know how your code is being used to draw the visualisation, then read on. Maybe it can help you figure out what is wrong if the visualisation doesn't work the way it should.

The most important method in GameBoard.java is the redraw method that gets called after every action by the player. There are five different components that are drawn by this method:

- the deck button (an instance of DeckTile)
- the score button (an instance of ScoreTile)
- the hand (multiple instances of LetterTile)
- the word (multiple instances of WordTile)
- the slots where the player can insert the letters (multiple instances of Slot)

Clicking on the deck button draws a letter tile from the deck to the hand by calling the draw method in Game.java, and you can see this in the following code in DeckTile.java

```
public void mouseReleased(MouseEvent e) {
  board.game.draw();
  board.redraw();
}
```

Clicking on the score button calls the scoreWord method in Game.java if there are three or more letters in the word. You can see this in the following code in ScoreTile.java:

```
public void mouseReleased(MouseEvent e) {
    if (_this.board.game.word.size() > 2) {
      _this.board.game.scoreWord();
    }
    _this.board.redraw();
  }
```

`ScoreTile.java` also display the score by directly accessing the `score` attribute in `Game.java` (line 56-58):

```
  g.setFont(new Font("SansSerif", Font.PLAIN, 40));
  String sc = this.board.game.score+"";
  g.drawString(sc,130-g.getFontMetrics().stringWidth(sc),52);
```

The hand is drawn using the following for loop in the method `redraw` in `GameBoard.java`

```
  for(int i = 0; i < game.hand.size(); i++) {
    this.add(new LetterTile(this,70+(i+1)*60,30,game.hand.get(i)));
  }
```

which as you can see calls the `get` and `size` methods in `Hand.java`. The logic for dragging a tile from the hand to the word (or more precisely the yellow slots) are written in the following code in `LetterTile.java`:

```
  public void mouseReleased(MouseEvent e) {
    ...
    // if the dragged tile is not the leftmost tile, send it back to the hand
    if (_this.board.game.hand.leftmost() != _this.letter) {
      _this.setBounds(startX,startY,width,height);
    }
    // else if the mouse pointer is on either the left or right slot,
    // move the current letter the the appropriate position on the word
    else if ( ... ){
      _this.board.game.moveFromHandToStartOfWord();
    }
    else if ( ... ){
      _this.board.game.moveFromHandToEndOfWord();
    }
    // in any other case, send it back
    else {
      _this.setBounds(startX,startY,width,height);
    }
    _this.board.redraw();
```

I removed parts of the code because I only want you to see that it uses the method `leftmost` from `Hand.java` (to check if the dragged tile is indeed the leftmost tile), and also the two methods `moveFromHandToStartOfWord` and `moveFromHandToEndOfWord` from `Game.java` to move the tiles from the hand to the word.

Drawing the word on the game board is done by calling the `toArray` method of `Word.java` and then creating a `WordTile` for each letter in the array (also in the `redraw` method of `GameBoard.java`):

```
  Letter[] w = game.word.toArray();
  int i = 0;
  while(i < w.length) {
    this.add(new WordTile(210+(i+1)*70,130,w[i]));
    i++;
  }
```

However, `WordTile.java` does not communicate with `Game.java`, although it uses the methods `getLetter` and `getScore` from `Letter.java`. The class `Slot.java` also do not use any classes in the `game package`, since it is only used as a marker for the word (the logic is actually controlled by `LetterTile.java` since the slot gets activated when we drag a letter tile to it).

I hope the above documentation helps you if you want to learn more about the code, or at least explains how your code is being used. Again, you do not need to know the details at all since your marks are determined by the JUnit test, but it is good to explore the code so that you have an idea of what it takes to actually build an application, even if it is a very simple one.

Marks and Grade Distribution

Your assignment is going to be marked out of 100, with 80 marks coming from the automarker and 20 marks from the quality of your code (i.e. coding style, to be handmarked). Aspects of the coding style to be marked in this assignment is similar to the ones assessed in the second assignment, so this includes your use of variable names, indentation, comments, and logic. For the automarking, here is the list of methods and tasks that you need to complete and their marks:

- Pass Level (51 marks):
  - in `Game.java`:
    - (3 marks) — `Game` (class constructor)
    - (4 marks) — `scoreWord`
  - in `Deck.java`:
    - (4 marks) — `Deck` (class constructor)
    - (4 marks) — `size`
    - (4 marks) — `add`
  - in `Hand.java`:
    - (4 marks) — `size`
    - (4 marks) — `add`
  - in `Word.java`:
    - (4 marks) — `size`
    - (4 marks) — `addToStart`
    - (4 marks) — `addToEnd`
    - (4 marks) — `isWord`
    - (4 marks) — `isPossibleWord`
    - (4 marks) — `getScore`
- Credit Level (8 marks):
  - in `Deck.java`:
    - (4 marks) — `remove`
  - in `Hand.java`:
    - (4 marks) — `remove`
- Distinction Level (8 marks):
  - in `Game.java`:
    - (4 marks) — `draw`, `moveFromHandToStartOfWord`, `moveFromHandToEndOfWord`
  - in `Hand.java`:
    - (4 marks) — `get`
- High Distinction Level (13 marks):
  - in `Word.java`:
    - (4 marks) — `toArray`
  - (4 marks) – efficient implementation of `add` method in `Deck.java` and `Word.java`
  - (5 marks) – efficient implementation of `size` method in `Deck.java`, `Hand.java`, and `Word.java`

List of methods to finish, organised by the class (note that another 9 marks come from the efficiency tests, so it is not listed below):

- in `Game.java` (11 marks):
  - (3 marks) — `Game` (class constructor) (Pass)
  - (4 marks) — `scoreWord` (Pass)
  - (4 marks) — `draw`, `moveFromHandToStartOfWord`, `moveFromHandToEndOfWord` (Distinction)
- in `Deck.java` (16 marks):
  - (4 marks) — `Deck` (class constructor) (Pass)
  - (4 marks) — `size` (Pass)
  - (4 marks) — `add` (Pass)
  - (4 marks) — `remove` (Credit)
- in `Hand.java` (16 marks):
  - (4 marks) — `size` (Pass)
  - (4 marks) — `add` (Pass)
  - (4 marks) — `remove` (Credit)
  - (4 marks) — `get` (Distinction)
- in `Word.java` (28 marks):
  - (4 marks) — `size` (Pass)
  - (4 marks) — `addToStart` (Pass)
  - (4 marks) — `addToEnd` (Pass)
  - (4 marks) — `isWord` (Pass)
  - (4 marks) — `isPossibleWord` (Pass)
  - (4 marks) — `getScore` (Pass)
  - (4 marks) — `toArray` (High Distinction)

Submission Format

You need to submit four files:

- `Game.java`
- `Deck.java`
- `Hand.java`
- `Word.java`

Please submit the files individually (do not zip them together or you may be penalised).

Penalties

Similar to the other assignments, there are some other strict conditions that you must follow:

- Failure to enter personal details (name and student number) in each of the submitted file will result in a penalty. You also need to tick the box the state that the assignment is your own work (if this doesn't show, Eclipse may be 'folding' (hiding) the lines, so please double check.
- Any compilation errors and/or infinite loops in any of the submitted files will result in an automatic zero.
- Submissions must be self-contained and does not require any other file besides `Letter.java` (and the JUnit test files). Any dependency on an external file (e.g. using functions from outside the submitted files) will result in an automatic zero. You must also not modify `Letter.java` since it is not part of your submission.
- Changing any package declaration in any submitted file will result in an automatic zero.
- Adding any import statement in any submitted file will result in an automatic zero.
- Modifying the visibility of any methods or attributes in any of the submitted file will result in an automatic zero (please leave everything as public so that the JUnit tests can access them directly).

Change Log

If there are any changes to this documentation after its release, it will be listed here. Please check to ensure you have read the latest documentation. Any changes will be announced on iLearn.