COMP1721 Object-Oriented Programming

Coursework 1: Creating & Using Classes

1 Introduction

Your task is to implement a Java version of the recently popular word game Wordle, using classes that follow our specifications. These classes are described by the UML diagram in Figure 1.

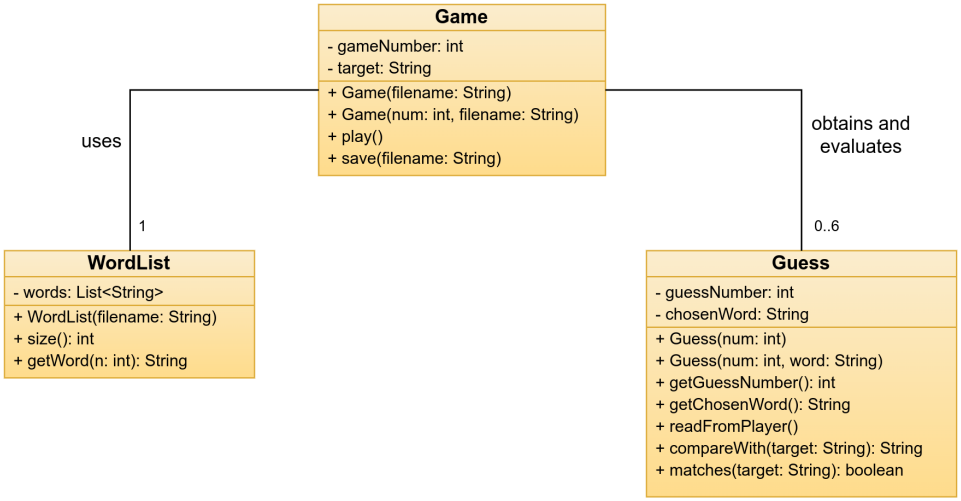

Figure 1: Main classes used in Coursework 1.

The class `WordList` represents the list of words used by Wordle. Words are loaded from a file, and the name of this file is supplied when creating a `WordList` object. The size of the word list can be obtained by calling the `size()` method on a `WordList` object. A word is chosen from the list using the `getWord()` method, passing in an integer representing the **game number**. A value of 0 means Game 0—i.e., the first ever game of Wordle. For this value, the method should return the first word in the list. For a value of 1, it should return the second word in the list, and so on.

The class `Guess` represents a single guess in a game of Wordle. A `Guess` object knows the **guess number**, an `int` value in the range 1–6. This value is stored in the `guessNumber` field. A `Guess` object also stores the word chosen by the player, in the `chosenWord` field. The constructors of `Guess` initialize either the guess number alone, or both guess number and chosen word. Validation is done for both fields.

If a `Guess` object is created without initializing the chosen word, then the `readFromPlayer()` method can be used to obtain that word from the player.

The `Guess` object provides two methods that can be used to evaluate a player's guess. The `matches()` method takes a string parameter representing the **target word** (the word the player is trying to guess). The method returns `true` if the player's chosen word matches the target, `false` if it doesn't match. The other method, `comparesWith()`, also takes the target word as a parameter. It returns a string in which ANSI escape codes are used to render the letters of the player's chosen word in different colours: green means the letter is in the target word and is in the right position; yellow means the letter is in the target but in the wrong position; and white means the letter is not in the target.

The class `Game` maintains associations with a single `WordList` object and a sequence of `Guess` objects. It also has fields to represent the game number and the target word associated with that game number. These fields are initialized in different ways by the two constructors. The differences are explained later.

The class provides two other methods. The `play()` method plays an entire game of Wordle. It will need to create and store `Guess` objects representing each guess made by the player. The method will also need to check each guess, printing the string returned by the `compareWith` method.

The `save()` method of `Game` has a string parameter representing a filename. The method should save a summary of the game to the specified file. This summary should consist of one line for each guess made. This line should contain the string returned by the `comparesWith()` method.

2 Preparation

It is important that you follow the instructions below *precisely*.

1. Download `cwk1-files.zip` from Minerva or Teams. **Put this file in the `coursework` directory of your repository**.
2. Unzip the Zip archive. You can do this from the command line in Linux, macOS and WSL 2 with `unzip cwk1-files.zip`.
3. Make sure that you have the following files and subdirectories immediately below `cwk1`:

   ```
   build.gradle   gradle/            gradlew.bat   settings.gradle
   config/        gradle.properties  README.html   src/
   data/          gradlew            README.md
   ```

   IMPORTANT: Make sure that this is exactly what you see! For example, you should NOT have a subdirectory of `cwk1` that is itself named `cwk1`. Thus the path to the README file, relative to the repository directory, should be `coursework/cwk1/README.md`. Fix any problems with the directory structure before proceeding any further.
4. Remove `cwk1-files.zip`. Use Git to add and commit the new files, then push your commit up to gitlab.com. The following commands, executed in a terminal window while in the `coursework` directory of your repository, will achieve all of this:

   ```
   git add cwk1
   git commit -m "Initial files for Coursework 1"
   git push
   ```

Important note for Windows Users

If you are doing this work on your own Windows PC, **make sure you have the Windows Terminal app installed**. You will need to use this to see meaningful output from the Wordle application when you run it. The standard command window that appears when you run `cmd.exe` will NOT be suitable.

You can obtain the Windows Terminal app from the Microsoft Store.

3 Class Skeletons

An essential first step is to create **skeletons** of the three classes from Figure 1. You need to write **stubs** (dummy versions) of the constructors and methods in these classes, to ensure that the tests compile and run. Once the tests are running successfully, you can repeatedly run them to check that you are implementing the classes properly.

The `.java` files for these classes are in `src/main/java/comp1721/cwk1`. Edit each of these files and implement stubs for the features shown on the UML diagram. Use the 'To Do' comments in each file and the notes below to guide you.

- Use identical method names to those shown on the diagram.
- Give your stubs the same return types as the methods shown on the diagram.
  - Use `void` for a method if there is no return type on the diagram.
  - Do not use `void` when defining a constructor!
- Stubs of constructors and `void` methods should be empty.
- Stubs for methods that return something should return a fixed value—e.g., zero for methods that return an `int`, or an empty string for those that are supposed to return a `String` object.

When you've implemented the stubs, run the tests from the command line like this:

```
./gradlew test
```

(On Microsoft Windows, omit the `./` from this command.)

If you've implemented the stubs correctly, the tests should compile and run, but you should see FAILED next to each of them—a total of 13 failures.

4 Basic Solution (13 marks)

For the basic solution, you need to implement the `WordList` and `Guess` class as specified in Figure 1. This part of the assignment is marked automatically, with your mark being based on the number of automated tests that pass. The subsections below provide implementation advice for each class.

4.1 Implementing `WordList`

You should find that Lectures 7 & 8 and Exercise 8 are helpful in showing you what is needed in the `WordList` class.

In the constructor, use one of the approaches discussed in Lecture 8 to read the words from the specified file. You should not catch any exceptions here, so you will need to write an exception specification indicating that an `IOException` can be thrown by the constructor.

The `getWord()` method should do some validation of the provided `gameNumber` parameter. It should throw a `GameException` if the value of this parameter is invalid.

4.2 Implementing `Guess`

The one-parameter constructor should initialize the `guessNumber` field from the supplied parameter. It should validate this parameter, checking to make sure that it is in the allowed range of 1–6, throwing a `GameException` if not. This constructor doesn't need to do anything to initialize `chosenWord`; Java will ensure that its value is `null`.

The two-parameter constructor should initialize the `guessNumber` and `chosenWord` fields from the supplied parameters. It should handle guess number the same way as the one-parameter constructor. It should validate the supplied value for the `chosenWord` field, checking that it consists of 5 alphabetic characters, throwing `GameException` if not. It should call `toUpperCase()` on the supplied string to ensure that `chosenWord` is in upper case.

The `compareWith()` method will be the most complex of the class. This method will need to build up the output string piece by piece. It will need to compare each letter of the chosen word with the corresponding letter of the target word. When the two letters are the same, it will need to output the letter with a green background, using ANSI escape codes. For example, if `C` is the first letter of both the chosen word and the target word, the following will need to be output:

```
"\033[30;102m C \033[0m"
```

If, on the other hand, the `C` of the chosen word occurs at a different location in the target word, a yellow background will be needed:

```
"\033[30;103m C \033[0m"
```

Finally, if there is no `C` in the target word, it should be given a white background:

```
"\033[30;107m C \033[0m"
```

5 Full Solution (9 marks)

For the full solution, you need to implement the `Game` class as shown in Figure 1. The following subsections provide implementation advice for this class.

Note that you should not implement a `main` method in `Game`. The `main` method has been provided for you, in a separate class named `Wordle`. You can study the code for this class if you like, but you should not modify this code in any way.

Once `Game` has been implemented properly, you should be able to run the Wordle application from the command line with

```
./gradlew run
```

This will run today's Wordle game—see Figure 2 for an example. If you want to run with a 'fixed' game, in which the target word is always the same, use this instead:

```
./gradlew runFixed
```

This will run Wordle 100 (for which the target word is BASIC).

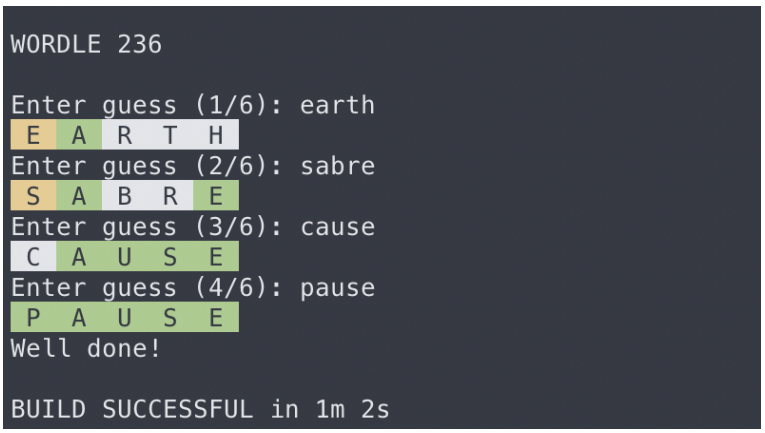

Figure 2: Example of `Wordle` program behaviour.

5.1 `Game` Constructors

The one-parameter constructor is used to create a `Game` object that plays today's game of Wordle. The parameter to this constructor is the name of the words file, which can be used to create a `WordList` object containing the required words.

In order to initialize the `gameNumber` field, the one-parameter constructor will need to determine the number of today's game. This value is simply the number of days that have elapsed since the very first Wordle game, Game 0, on 19 June 2021.

Computing the number of days can be accomplished fairly easily using the `LocalDate` class, from the `java.time` package. For example, you can create a `LocalDate` set to a specific date such as 10 February 2022 by using

```
LocalDate.of(2022, 2, 10)
```

You can also create a `LocalDate` set to today's date using `LocalDate.now()`. Given two of these date objects, there are several different ways of determining the time interval between them—see Baeldung's Introduction to the Java Date/Time API for examples.

Once the game number has been determined, this can be used together with the previously created `WordList` object to get today's target word.

The two-parameter constructor is a bit simpler because it uses the game number given to it as a parameter, rather than determining the game number for today's game.

Both constructors will need an exception specification for `IOException`. Do not try to catch exceptions of this type inside the constructors!

5.2 The `play()` Method

The `play()` method should play a game of Wordle, with player input and program output following the example in Figure 2. You will need to create `Guess` objects and store them, for later use in the `save()` method (see 5.3). Each `Guess` object will need to be loaded with the word chosen by the player, after which a comparison should be made with the target word, the results being printed in the terminal.

At the end of a successful game, the application should display one of three possible outputs, as shown in Table 1. If the player has failed to guess the target word, the application should display the message `Nope - Better luck next time!` on one line, followed by the target word on the next line.

Guesses	Expected output
1	`Superb - Got it in one!`
2–5	`Well done!`
6	`That was a close call!`

Table 1: Possible outputs from a successful game.

5.3 The `save()` Method

The `save()` method should save a game summary to a text file with the given filename. Essentially, this means printing to a file the strings returned when the `compareWith()` method is called on the `Guess` objects that were previously stored during execution of the `play()` method.

The easiest approach here is to create a `PrintWriter` object associated with the given file—see Lecture 8 for examples. Once you've done this, you can call `println` on that object to write lines to the file.

The `Wordle` program is configured to call `save()` with `build/lastgame.txt` specified as the output file. You can examine this file in the terminal to see whether your implementation functions correctly. Figure 3 shows an example of doing this.

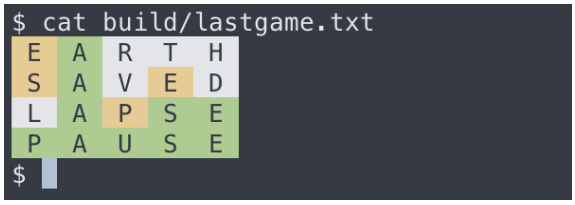

Figure 3: Examining the file generated by the `save()` method.

On Windows, which doesn't have a `cat` command, you can use `type build\lastgame.txt` instead.

6 Advanced Solution (6 marks)

The following subsections describe optional features that you can implement to earn a few extra marks. You can tackle as many of these as you like, but the maximum reward for advanced features is 6 marks.

Some of the options listed here are considerably more challenging. You should attempt them only if you manage to complete the Basic and Full solutions fairly quickly and easily.

6.1 Accessible Wordle

Wordle isn't particularly accessible to people who have impaired colour vision, or people who rely on screen readers. Add an accessibilty option to the application so that it can display more accessible output. The option should be activated by supplying `-a` as the first command line argument. It should still be possible to specify the game number as an additional command line argument.

With the accessibility option enabled, the example of Figure 2 could look like this:

```
Enter guess (1/6): earth
1st correct but in wrong place, 2nd perfect
Enter guess (2/6): sabre
1st correct but in wrong place, 2nd and 5th perfect
Enter guess (3/6): cause
2nd, 3rd, 4th and 5th perfect
Enter guess (4/6): pause
You won!
```

Implementing this feature will earn up to 2 additional marks.

Note that you'll need to modify `build.gradle` here, adding a new run task that runs the application with the `-a` option. You can copy and alter the `runFixed` task for this.

6.2 History and Summary Statistics

Add a history feature to the application, so that it stores the outcome of each game in a file named `history.txt`. Information stored in this file should include the game number, whether the word was guessed successfully or not, and the number of guesses that were made.

Then used the saved history to implement the equivalent in the terminal of Wordle's summary screen. At the end of play, the application should display

- Number of games played

- Percentage of games that were wins
- Length of the current winning streak
- Longest winning streak
- Histogram of guess distribution

Implementing both of these features will earn up to 4 marks.

6.3 JavaFX GUI

Investigate **JavaFX**, a framework for developing GUIs in Java. Use JavaFX to create a separate GUI-based version of the Wordle application. Reuse the existing `WordList` and `Guess` classes for this. (You can add extra methods to these classes if needed, but don't remove or change any of the existing methods.)

Note: you will need to make some changes to the `build.gradle` file to support JavaFX, as documented on the OpenJFX website. You'll also need to add another run task to the file.

Warning: This will be challenging and will require substantial investigation into JavaFX. Do NOT attempt this if you are short on time!

Implementing the GUI will earn up to 6 marks.

7 Submission

Make sure that any relevant code changes have been committed to your Git repository and then pushed up to GitLab. Then generate a Zip archive of your solution using

```
./gradlew submission
```

This will create a file named `cwk1.zip`. Submit this file to Minerva, via the link provided in the *Submit My Work* section.

The deadline for submissions is **10 am on Friday 4 March**.

Note that all submissions will be subject to automated plagiarism checking.

8 Marking

13	Basic solution
9	Additional code for Full solution
6	Additional code for Advanced solution
4	Coding style and comments
3	Use of version control
35	