ACS-2947-002
Assignment 2
Due by Friday, March 4, 11:59 pm

Instructions

- Submit your `.java` files (together in a `Assign2.zip` file) via Nexus.
- Include your name and student number as a comment in every file.
    - Document the classes using Javadoc notation.
    - Include comments as needed.
    - Use appropriate exception handling where necessary.

PART A: (25 marks)

Write a simple text-based version of the classic board game Mastermind, where a single player is the code breaker, and the system is the code maker. The system selects a code of four coloured pegs and the player tries to guess the secret code.

In each round, the player makes a guess, and the system tells the player how many pegs of the guess were *exact* matches to the code (correct in both color and position, marked `'x'`), and how many colours were *partial* matches to the code (correct color placed in the wrong position, marked `'o'`). The feedback is displayed in a 2x2 grid format similar to the board game.

e.g., suppose the code is *black red blue green*

```
Guess #1:
blue red green yellow
x o
o -
```

The feedback shows that there is one exact match and 2 partial matches. Notice that this configuration does not indicate which pegs are exact matches. The player makes guesses until a) the player breaks the code (player wins!) or b) 12 guesses are made but did not result in a full match (system wins).

1. Create the generic `ArrayList` class that implements the provided `List` interface (note that `List` extends `Iterable`)
    a. Overload the `add` method: include another `add` method that will have one parameter: an element that adds to the *end* of the list.
    b. Override the `equals` method that checks if the `ArrayList` is equivalent to the given instance.
    c. Make your `ArrayList` dynamic: the array should grow to double its current capacity if it runs out of space and shrink to half its current capacity when the number of elements in the array goes below *N*/4, where N is the current capacity. Modify `add()` and `remove()` methods and include a `resize()` method to support the dynamic structure. Set the default capacity to 4.

√ 2. Create a class named `Peg` with field `colour`. Include any other fields/methods to help with gameplay. Override the `equals` method to return true if the colours match.

√ 3. Write a `MasterMind` class that acts as the code maker and handles the mechanics of the Mastermind game. Include a minimal main method that instantiates the game and invokes gameplay.

√ 4. In the main method, also illustrate how the capacity of your array would changes as objects are added and removed.

Your program should have the following:

√ a. An instance of `ArrayList` that holds a set of 4 pegs of which colours are randomly generated. Each peg has a colour of 6 different possibilities (duplicates are allowed, blanks are not).

√ b. Another `ArrayList` that holds pegs that represent the player's guess.

√ c. A game loop that prompts the user for their guess and determines if the 2 ArrayLists are equal:

i. if so, notify the player and end the game

ii. if not, provide the user feedback on their guess:

- Determine whether if each peg of the guess is a match and mark it accordingly. You will need to compare the guess against the code and determine the number of exact and partial matches.

remove the checked element?

√ d. After their 12th guess, if it is not a full match, inform the player that the system won.

Note:

- You may assume that the player input is valid i.e., if it is an invalid colour, the player loses and the game is over.
- Enums are optional (colour, match status)

Suggestions:

- Display the generated code for testing (and remove before submitting)
- For guess feedback: must be careful to avoid counting any of the pegs twice; make at least two passes to compare the guess and the code. In the first pass, look for exact matches and in the second pass, look for partial matches.

Sample output:

[code: white blue yellow green]

```
System:     Guess #1:
Player:     blue blue blue blue
System:     x -
            - -

System:     Guess #2:
Player:     blue red red red
System:     o -
            - -
```

```
System:     Guess #3:
Player:     yellow blue yellow yellow
System:     x x
            - -

System:     Guess #4:
Player:     green blue yellow green
System:     x x
            x -

System:     Guess #5:
Player:     green blue yellow black
System:     x x
            o -

System:     Guess #6:
Player:     white blue yellow green
System:     You cracked the code!
```

PART B: ArrayPositionalLists (55 marks)

Implement a Positional List using an **array.** Refer to page 281 in your textbook.

√ 1. Create two classes called `ArrayPositionalList` (APL) and `ArrPos` (nested class in APL) that implement the provided `PositionalList` and `Position` interfaces, respectively. Note that `PositionalList` extends `Iterable`.

√ 2. Demonstrate your implementation by rewriting the Scoreboard example from L02 . Use a `PositionalList` for the board, and in a `GameDriver` class use the players from the notes to demo (build the scoreboard in any way then add Jill and remove Paul). Illustrate all other implemented public methods in the driver class.

To implement the `ArrayPositionalList`, use both the `LinkedPositionalList` and `ArrayList` implementations as a guide.

√ a) Add the nested ArrPos class. `ArrPos` implements the `Position` interface (just like `Node` in a *linked* positional list). Note that there is no `next` or `prev`, but only an integer `index` and generic `element`.

√ b) Add the fields and constructors:
- an array of `ArrPos` objects
- a constant `CAPACITY` defaulted at 16, size field
- two constructors: no-arg and capacity as a parameter

√ c) Add your `size()` and `isEmpty()` methods.

√ d) Implement the `first()` and `last()` methods: how would you get the first and last elements from the array? This should form a basis of how to move from linked to array. From there you can start thinking about how to convert all the methods from linked-based to array-based implementation.

√ e) Add both a `PositionIterator` and an `ElementIterator` that support the iteration of positions and elements, respectively. Refer to your text/notes (L09_) for necessary nested classes and methods

- `PositionIterator, hasNext(), next(), remove()`
- `ElementIterator, iterator(), positions()`

*Note that you require the `java.util.Iterator`

√ f) Use the enhanced for loop in the implementation of `toString` in `Scoreboard` class.

To consider: with `LinkedPositionalList`, you get the previous and next positions through the node (which is a `Position`) and `getNext()` and `getPrev()` methods. With `ArrayPositionalList` you get the next and previous through the `Position` as well, but with the `getIndex()` method and the array. Instead of simply calling `node.next()` you will find out what `ArrPos.getIndex()` is, then return the `ArrPos` at the *next index* of your array.

Note:

- You will need a way to validate and explicitly cast `Position` objects to `ArrPos` objects in order to use `ArrPos` methods like `getIndex()`
- Many methods declare exceptions in its signature: most can be handled in common private utility methods
- The `index` field of the `ArrPos` class needs to be updated with any methods that require a *shift* in elements

Suggestions:

- Override the `toString` method to display both index and element. For example, an `ArrayPositionalList` populated with names will be displayed as:
  `[0] Bob [1] Alice [2] Simon [3] Theodore [4] Alvin [5] David`
  - this can be useful for testing/debugging
- Test each individual method as you write it.

Submission

Submit your **`Assign2.zip`** file that include all the assignment files (`List.java`, `ArrayList.java`, `Peg.java`, `MasterMind.java`, `Position.java`, `PositionalList.java`, `ArrayPositionalList.java`, `Scoreboard.java`, `GameEntry.java`, `GameDriver.java`, any enum or other class used) via **Nexus**.