Java exercises 2

Please see the separate sheet about running Eclipse if you are not familiar with the Eclipse IDE.

Objectives

- Writing more complex Java classes, using enumerated types, collections, and other data structures
- Writing unit tests
- Implementing simple GUI actions using Swing
- Using online documentation to understand the details of built-in Java library methods

Task 1

You must develop a set of classes to allow users to play **Rock-Paper-Scissors** (referred to from now on as **RPS**) against a computer opponent. If you are not familiar with the rules of the game, they can be found at https://en.wikipedia.org/wiki/Rock–paper–scissors.

At a high level, your code should be structured as follows:

- Two enumerated types, **Symbol** and **GameResult**, representing the list of possible symbols that can be used in the game and the list of possible outcomes of a single game.
- A **GamePlayer** class representing a single (computer) player in the game.
- A **GameMain** class containing a **main** method to run the game.

Symbol and GameResult

GameResult should have three possible values: **WIN**, **LOSE**, and **DRAW** – these represent the three possible outcomes of a single game of RPS.

The **Symbol** type represents the three possible symbols in the game: **ROCK**, **PAPER**, and **SCISSORS**. In addition to those three values, you should also implement an instance method **getResult()** inside **Symbol** that compares the current symbol to another and returns one of the three possible **GameResult** values, using the following rules.

- For any **Symbol** value s, **s.getResult(s)** should return **GameResult.DRAW**
- Otherwise, the result should be selected based on the following rules:
    - Scissors defeats Paper
    - Paper defeats Rock
    - Rock defeats Scissors
- The result should indicate the result for the **first** symbol; that is, **ROCK.getResult(SCISSORS)** should return **GameResult.WIN**, while **SCISSORS.getResult(ROCK)** should return **GameResult.LOSE**.

GamePlayer

This class must represent a computer playing the game of RPSLS. It should provide a single instance method, **getSymbol()**, that returns the next symbol to be used in the game. You can use whatever strategy you want to choose the symbol – the easiest method is to choose at random (see https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Random.html for the documentation of the **java.util.Random** class). If you want, you can also try adding extra fields and methods to the class to base the choice on the human's recent choices as in the well-known strategies listed at https://en.wikipedia.org/wiki/Rock%E2%80%93paper%E2%80%93scissors#Algorithms.

GameMain

This class should have a single **main** method that proceeds as follows:

- Initialise a **GamePlayer** instance
- Prompt the user for the number of rounds to play (you can use the **java.util.Scanner** class to read input from the user, documentation for this class is at https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Scanner.html)
- For each round:
    - Prompt the user for their chosen Symbol
    - Choose a symbol for the computer player using your **getSymbol()** method
    - Figure out the result of the round using **Symbol.getResult()**, print the result, and keep track of the wins for each player
- At the end of all rounds, print the total score for each player (computer and human)

Task 2: Bin Packing

You are to design and implement a set of classes that will assign items to bins based on the weights of the items and the capacity of the bins. Each bin has a fixed (identical) **capacity**, and each item has a **weight**. The problem is to take a list of item weights and choose the minimum set of bins that will contain the items. This is a well-known problem from computer science known as **bin-packing** – you can read more about the problem at https://en.wikipedia.org/wiki/Bin_packing_problem.

We will use several approximate algorithms:

- **NextFit**, which places each item in the last bin that was used, or else in a new bin if it is too heavy for that bin.
- **FirstFit**, which places each item in the first non-empty bin that will accommodate it, or else a new bin if it is too heavy for any existing bin.
- **BestFit**, which places an item in a bin whose spare capacity is as close as possible to the weight of the item, or in a new bin if it is too heavy for any existing bin.

Consider the following sequence of values, with a bin capacity of 100:

- 75 50 20 60 40 50

NextFit would proceed as follows:

1. 75 goes into newly-created bin 0
2. 50 does not fit into bin 0, so goes into newly-created bin 1
3. 20 goes into bin 1 as well
4. 60 does not fit into bin 1, so goes into newly-created bin 2
5. 40 goes into bin 2 as well
6. 50 does not fit into bin 2, so goes into newly-created bin 3

The resulting bins have loads (75, 70, 100, 50).

FirstFit would proceed as follows:

1. 75 goes into newly-created bin 0
2. 50 does not fit in any existing bin, so goes into newly-created bin 1
3. 20 fits into bin 0, so goes in there are well
4. 60 does not fit into any existing bin, so goes into newly-created bin 2
5. 40 fits into bin 1, so goes in there as well
6. 50 goes into newly-created bin 3

The resulting bins have loads (95, 90, 60, 50).

BestFit would proceed as follows:

1. 75 goes into newly-created bin 0
2. 50 does not fit into any existing bin, so goes into newly-created bin 1
3. 20 fits most closely into bin 0 (free space 25), so goes there
4. 60 does not fit into any existing bin, so goes into newly-created bin 2
5. 40 fits most closely into bin 2 (free space 0), so goes there

6. 50 fits most closely into bin 1, so goes there

The resulting bins have loads (95, 100, 100).

For this set of data, the best-fit algorithm wins, in the sense that it uses fewest bins, i.e. 3 as compared to the 4 crates required by the other two algorithms. However, it is also true that the best-fit algorithm is doing the most work to solve the problem, as it must consider every bin for every item, while next-fit must consider only the most recent bin and first-fit only needs to continue until it finds any bin where the weight fits.

Bin and PackingStrategy

You must first create a **Bin** class with two fields: an integer **capacity**, and a list of integers representing the **weights** stored in the bin. The Bin constructor should take a single argument, an **int** representing the capacity. You should also implement the following methods:

- **public void store(int weight) throws IllegalArgumentException**: adds the given value to the list inside the bin, or throws an **IllegalArgumentException** if the weight cannot fit
- **public int getSpace()**: returns the remaining space in the **Bin** based on the total weight of objects included

You must then create a **PackingStrategy** interface which contains one static method, **pack**, with the following signature:

public Set<Bin> pack(int capacity, List<Integer> values);

Implementing the algorithms

Create three classes implementing the above three packing strategies. Each class should implement the **PackingStrategy** interface and should implement one of the packing strategies above. In all cases, the **pack** method should process the given list of item weights and should return a **Set** of **Bin** objects of the given capacity, following the procedure for each packing strategy. **None of the methods should modify the input list of values in any way**.

The class names should be **NextFitStrategy**, **FirstFitStrategy**, and **BestFitStrategy**, respectively.

Task 3: Unit testing

For both of the programs implemented above, define **unit tests** to test the behaviour of your implemented methods – good candidates are **Symbol.getResult()** and the **pack()** methods of the three packing classes. Here is an overview of how to create new tests in Eclipse:

- Right click on the project and choose **New – JUnit Test case**
- Tick **New JUnit Jupiter test** at the top, put the class name into the **Class under test** field (e.g., put **Symbol** if you are writing test cases for **Symbol**), and give your test class a name (e.g., for testing Symbol, you should probably call your test class **TestSymbol**).
- In the next screen, tick the methods that you plan to test and then press **Finish**
- You might be prompted to add the JUnit 5 library to the build path – you should do that.

Then you should fill in the body of all test methods. For testing **Symbol.getResult()**, for example, you might want to define several tests, including:

- Test whether it correctly returns **GameResult.DRAW** when a symbol is compared against itself
- Test whether it correctly returns **WIN** and **LOSE** in the correct situations as listed above

When testing the **pack()** methods of the packing strategies, you can see whether the output of each method is as specified above on the given input list; you can also test the behaviour on other inputs, for example an empty list.

Task 4: Swing programming

Using the sample Swing program from the lecture slides as a starting point, add a listener to the JButton to produce different behaviour the user clicks the button. For example:

- Make the window close
- Pop up a dialogue box asking whether the user really wants to close the window, and only close it if they press **OK** in the dialogue box
    - Here the documentation of **JOptionPane** will be useful -- https://docs.oracle.com/en/java/javase/12/docs/api/java.desktop/javax/swing/JOptionPane.html
- … any other behaviour you can think of!