Activity 11: Simulation of Elevens (Optional)

Introduction:

We have implemented two different solitaire games that we can play, Elevens and Thirteens. That is perfect if we want to entertain ourselves playing them. But what if we want to answer questions about the games? For example, what percentage of Elevens games can be won? You probably already have some idea about the answer to this question, but you might have to play thousands of games to have any real confidence in your answer. That's where *simulation* comes in.

Exploration:

A common computing application is the *simulation* of some process; in other words, writing a program that imitates the process in some way. The behaviors and state in the program represent key features of, and are said to provide a *model* for, the process. An example is a simulation of a household robot vacuum cleaner. Internally, the simulation program is keeping track of its environment, the battery level, and the amount of dust in its dust container, and is perhaps displaying them on a computer console. Program methods plot the robot's course through rooms in the "house" and determine when it is finished.

A simulation is useful when the process being simulated is too complicated, too slow, too dangerous, or too expensive to observe in the real world. Also, understanding the program helps one understand the process being simulated. For example, the producers of the robot vacuum cleaner would have used a simulation to debug its algorithms before starting to manufacture the actual robots.

A program is called *probabilistic* when its state change is affected by chance. One example is a traffic simulation, which has to account for cars unpredictably entering the traffic zone and driving at varying speeds. A more obvious example is a simulation of a game based on dice or spinners. In Elevens, the probabilistic element is the shuffling of the deck of cards.

To model random events, we use a *pseudo-random number generator* (the "pseudo" is usually omitted). In the `Deck shuffle` method, we used the `Math.random` method to generate our random numbers. This reliance on chance significantly complicates the task of verifying that a simulation is behaving correctly. The programmer needs to have a good idea of what output to expect. Also, a small number of outcomes of the chance events may produce misleading behavior. For example, four flips of a coin may produce all heads, but it would be a mistake to assume that this behavior would happen often. Ten thousand flips would produce the more reasonable outcome of around 50 percent heads and 50 percent tails. The typical probabilistic simulation involves a large number of calls to the random number generator to increase the likelihood that the output reflects expected behavior.

To simulate Elevens, we will need to model the "playing" of the game using program state and behavior. Let's see how the real world relates to the code:

STATE:

Real-world "data"	Program data
The deck of cards	A `List` of `Card` objects
The cards on the board	An array of `Card` objects

BEHAVIOR:

Real-world operations	Program operations
Locating cards to remove	Searching for specific groups of `Card` objects in the board (an 11-pair or a JQK-triplet)
Removing cards and replacing them	Removing `Card` objects from the board and replacing them with `Card` objects from the deck
Dealing a card	Removing a `Card` object from the deck

We have most of this code already written. We have already taken care of all the state requirements. The deck of cards is modeled by the `cards` list in the `Deck` class. And the cards on the board are represented by the `cards` array in the `Board` class. We have also written methods for most of the necessary behaviors. In fact, we only need to model the additional things you do when you are playing the game yourself!

In the exercises, you will use an `ElevensSimulation` class, which will play games of Elevens. It will need access to methods that mimic the actions you make when you play the game. What do you do when you play the game and why do you do it? Answer these questions:

- What do you do repeatedly to play a game?
- As you are scanning the cards on the board, what are you trying to find?
- Why do you decide to click on a group of cards?
- What happens when you click the **Replace** button?

We will model these behaviors with three new methods in the `ElevensBoard` class:

- `playIfPossible` — Looks for a legal play and makes the play (if found). This is the only new method that `ElevensSimulation` needs to call directly. We could put all of our new code into this method, but it's helpful to divide it up using two new `private` helper methods.
- `playPairSum11IfPossible` — Looks for an 11-pair and replaces it (if found).
- `playJQKIfPossible` — Looks for a JQK-triplet and replaces it (if found).

Next we consider the implementation of `playPairSum11IfPossible`. This method needs to first determine if the board contains an 11-pair. Then, if it finds one, it needs to remove it. Of course, `playPairSum11IfPossible` could call `containsPairSum11` to see if there is an 11-pair on the board. But `containsPairSum11` doesn't return any information about the indexes of the two cards that make up the 11-pair. So, `playPairSum11IfPossible` would have to find the pair again before removing it. To avoid having to find the pair twice, we would need to copy the code from `containsPairSum11` into `playPairSum11IfPossible`. Thankfully, there's a better way.

What if we change the `containsPairSum11` method into a `findPairSum11` method? In other words, instead of having a "contains" method that returns a `boolean` value, we have a "find" method that returns a list of the indexes of the two cards in the pair. If there is no 11-pair on the board, it will return an empty list. Now, `playPairSum11IfPossible` will be able to call `findPairSum11`, and if there is an 11-pair, it will already have the list of indexes needed to call the `replaceSelectedCards` method. This design eliminates both the duplicated code and the double work! We will need to make a similar modification to change the `containsJQK` method into a `findJQK` method for the `playJQKIfPossible` method to call.

Note that it's usually not possible to foresee everything during the initial design of a program. For example, in the GUI version of Elevens, there was no need for "find" methods. The person playing the game had that task. However, the "find" methods became useful when we got into the details of the simulation. So, program designs can and do change.

Of course, when we do an initial program design, we try to accommodate all the needs of the problem as we understand them. But we also try to keep the design flexible, so that we can accommodate future needs. One rule of program design is that methods should be `private`, unless there is a good reason for another class to call those methods. Because we initially made `containsPairSum11 private`, we know that no other class uses it. Therefore, it's safe to rename and change it.

Exercises:

1. First, examine the completed `ElevensSimulation` class in the **Activity11 Starter Code** folder. This simulation creates an `ElevensBoard` object, and uses it to play `GAMES_TO_PLAY` games of Elevens. Note that the only new `ElevensBoard` method used is `playIfPossible`.

2. Now make the necessary changes to the `ElevensBoard` class. Change `containsPairSum11` into `findPairSum11`. You will need to change both the method heading and the method body. Note that the method's comment block has already been changed for you.

3. Change the `isLegal` and `anotherPlayIsPossible` methods to use `findPairSum11` instead of `containsPairSum11`. Note that the board contains an 11-pair if and only if `findPairSum11(cIndexes).size() > 0`.

4. Change `containsJQK` to `findJQK` in a similar fashion to the `containsPairSum11` to `findPairSum11` conversion you did in exercise 2 above. Again, the method's comment block has already been changed for you.

5. Change the `isLegal` and `anotherPlayIsPossible` methods to use `findJQK` instead of `containsJQK`. At this point, the Elevens GUI program should work just as it did before.

6. It's time to complete the `ElevensBoard` methods required by the `ElevensSimulation` class. First complete the `public playIfPossible` method, which is called from the `ElevensSimulation` class. This method will use the `private playPairSum11IfPossible` and `playJQKIfPossible` helper methods. Note that you will have to replace the `return` statement in the stubbed out method.

7. Complete the `private playPairSum11IfPossible` and `playJQKIfPossible` helper methods. Note that you will have to replace the `return` statements in the stubbed out methods.

8. Now it's time to test your simulation-related changes. Make sure that `GAMES_TO_PLAY` and `I_AM_DEBUGGING` are initialized to `1` and `true` respectively in `ElevensSimulation`. Also make sure that `I_AM_DEBUGGING` is initialized to `true` in `ElevensBoard`. Run the `ElevensSimulation` program a few times and examine the output. You should be able to see both 11-pairs and JQK-triplets being correctly identified and removed.

Questions:

1. Set the `I_AM_DEBUGGING` flags to `false` and `GAMES_TO_PLAY` to `10`. Run the `ElevensSimulation` program a few times and record the percentage of games won for each run. What is the range of win percentages that you saw? Were the percentages fairly consistent, or did they vary quite a bit?

2. Increase the number of games to play to `100`. Are the win percentages more consistent from run to run?

3. Experiment with simulating different numbers of games. How many games do you need to play in order to get consistent results from run to run?

4. Optional — Repeat the above steps for the Thirteens game.