Activity 8: Using an Abstract Board Class

Introduction:

The Elevens game belongs to a set of related solitaire games. In this activity you will learn about some of these related games. Then you will see how inheritance can be used to reuse the code that is common to all of these games without rewriting it.

Exploration: Related Games

Thirteens

A game related to Elevens, called *Thirteens*, uses a 10-card board. Ace, 2, … , 10, jack, queen correspond to the point values of 1, 2, …, 10, 11, 12. Pairs of cards whose point values add up to 13 are selected and removed. Kings are selected and removed singly. Chances of winning are claimed to be about 1 out of 2.

Tens

Another relative of Elevens, called *Tens*, uses a 13-card board. Pairs of cards whose point values add to 10 are selected and removed, as are quartets of kings, queens, jacks, and tens, all of the same rank (for example, K♠, K♥, K♦, and K♣). Chances of winning are claimed to be about 1 in 8 games.

Exploration: Abstract Classes

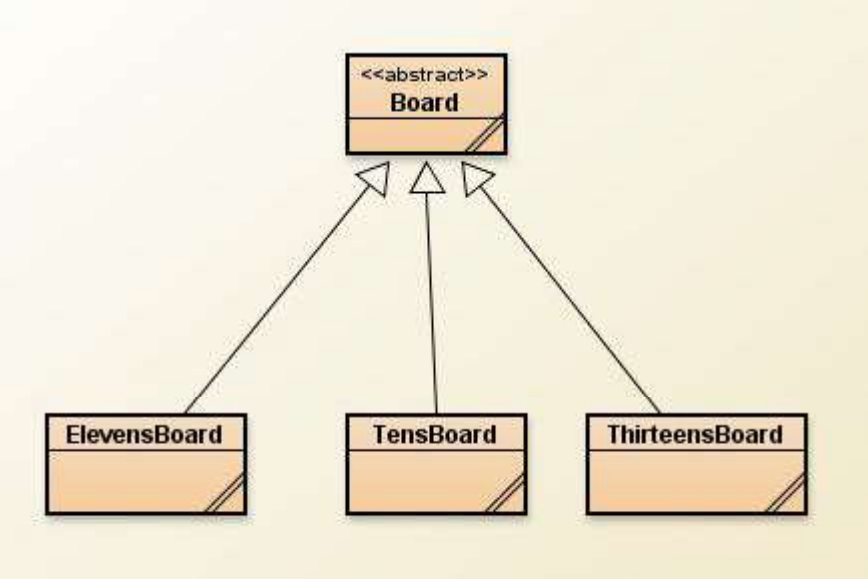
In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- State (instance variables) — a deck of cards and the cards "on the" board.
- Behavior (methods) — to deal the cards, to remove and replace selected cards, to check for a win, to check if selected cards satisfy the rules of the game, to see if there are more legal selections available, and so on.

With all of this state and behavior in common, it would seem that inheritance could allow us to write code once and reuse it, instead of having to copy it for each different game.

But how? If we use the "IS-A" test, a `ThirteensBoard` "IS-A" `ElevensBoard` is not true. They have a lot in common, but an inheritance relationship between the two does not exist. So how do we create an inheritance hierarchy to take advantage of the commonalities between these two related boards?

The answer is to use a common superclass. Take all the state and behavior that these boards have in common and put them into a new `Board` class. Then have `ElevensBoard`, `TensBoard`, and `ThirteensBoard` inherit from the `Board` class. This makes sense because each of them is just a different kind of board. An `ElevensBoard` "IS-A" `Board`, a `ThirteensBoard` "IS-A" `Board`, and a `TensBoard` "IS-A" `Board`. A diagram that shows the inheritance relationships of these classes is included below. Note that `Board` is shown as abstract. We'll discuss why later.



Let's see how this works out for dividing up our original `ElevensBoard` code from Activity 7. Because all these games need a deck and the cards on the board, all of the instance variables can go into `Board`. Some methods, like `deal`, will work the same for every game, so they should be in `Board` too. Methods like `containsJQK` are Elevens-specific and should be in `ElevensBoard`. So far, so good.

But what should we do with the `isLegal` and `anotherPlayIsPossible` methods? Every Elevens-related game will have both of these methods, but they need to work differently for each different game. That's exactly why Java has `abstract` methods. Because each of these games needs `isLegal` and `anotherPlayIsPossible` methods, we include those methods in `Board`. However, because the implementation of these methods depends on the specific game, we make them `abstract` in `Board` and don't include their implementations there. Also, because `Board` now contains `abstract` methods, it must also be specified as `abstract`. Finally, we override each of these `abstract` methods in the subclasses to implement their specific behavior for that game.

But if we have to implement `isLegal` and `anotherPlayIsPossible` in each game-specific board class, why do we need to have the `abstract` methods in `Board`? Consider a class the uses a board, such as the GUI program you used in Activity 6. Such a class is called a *client* of the `Board` class.

The GUI program does not actually need to know what kind of a game it is displaying! It only knows that the board that was provided "IS-A" `Board`, and it only "knows" about the methods in the `Board` class. The GUI program is only able to call `isLegal` and `anotherPlayIsPossible` because they are included in `Board`.

Finally, we need to understand how the GUI program is able to execute the correct `isLegal` and `anotherPlayIsPossible` methods. When the GUI program starts, it is provided an object of a class that inherits from `Board`. If you want to play Elevens, you provide an `ElevensBoard` object. If you want to play Tens, you provide a `TensBoard` object. So, when the GUI program uses that object to call `isLegal` or `anotherPlayIsPossible`, it automatically uses the method implementation included in that particular object. This is known as *polymorphism*.

Questions:

1. Discuss the similarities and differences between *Elevens*, *Thirteens*, and *Tens*.

2. As discussed previously, all of the instance variables are declared in the `Board` class. But it is the `ElevensBoard` class that "knows" the board size, and the ranks, suits, and point values of the cards in the deck. How do the `Board` instance variables get initialized with the `ElevensBoard` values? What is the exact mechanism?

3. Now examine the files `Board.java`, and `ElevensBoard.java`, found in the **Activity8 Starter Code** directory. Identify the `abstract` methods in `Board.java`. See how these methods are implemented in `ElevensBoard`. Do they cover all the differences between *Elevens*, *Thirteens*, and *Tens* as discussed in question 1? Why or why not?