AP® Computer Science A Elevens Lab Student Guide

The AP Program wishes to acknowledge and thank the following individuals for their contributions in developing this lab and the accompanying documentation.
Michael Clancy: University of California at Berkeley
Robert Glen Martin: School for the Talented and Gifted in Dallas, TX
Judith Hromcik: School for the Talented and Gifted in Dallas, TX

Activity 6: Playing Elevens

Introduction:

In this activity, the game Elevens will be explained, and you will play an interactive version of the game.

Exploration:

The solitaire game of Elevens uses a deck of 52 cards, with ranks A (ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (king), and suits ♣ (clubs), ♦ (diamonds), ♥ (hearts), and ♠ (spades). Here is how it is played.

1. The deck is shuffled, and nine cards are dealt "face up" from the deck to the board.
2. Then the following sequence of steps is repeated:
   a. The player removes each pair of cards (A, 2, … , 10) that total 11, e.g., an 8 and a 3, or a 10 and an A. An ace is worth 1, and suits are ignored when determining cards to remove.
   b. Any triplet consisting of a J, a Q, and a K is also removed by the player. Suits are also ignored when determining which cards to remove.
   c. Cards are dealt from the deck if possible to replace the cards just removed.

The game is won when the deck is empty and no cards remain on the table. Here's a sample game, in which underlined cards are replacements from the deck.

Cards on the Table									**Explanation**
K♠	10♦	J♣	2♣	2♥	9♦	3♥	5♠	5♦	initial deal
K♠	10♦	J♣	<u>7♦</u>	2♥	<u>Q♠</u>	3♥	5♠	5♦	remove 2♣ (either 2 would work) and 9♦
<u>A♠</u>	10♦	<u>9♣</u>	7♦	2♥	<u>7♣</u>	3♥	5♠	5♦	remove J♣ Q♠ K♠
A♠	10♦	<u>10♠</u>	7♦	<u>3♣</u>	7♣	3♥	5♠	5♦	remove 9♣ and 2♥ (removing A♠ and 10♦ would have been legal here too)
<u>2♠</u>	10♦	<u>9♠</u>	7♦	3♣	7♣	3♥	5♠	5♦	remove A♠ and 10♠ (10♦ could have been removed instead)
<u>A♣</u>	10♦	<u>K♦</u>	7♦	3♣	7♣	3♥	5♠	5♦	remove 2♠ and 9♠
<u>6♦</u>	<u>K♣</u>	K♦	7♦	3♣	7♣	3♥	5♠	5♦	remove A♣ and 10♦

2♦	K♣ K♦ 7♦ 3♣ 7♣ 3♥ 5♠ Q♦	remove 6♦ and one of the 5s; no further plays are possible; game is lost.

An interactive GUI version of Elevens allows one to play by clicking card images and buttons rather than by handling actual cards. When `Elevens.jar` is run, the cards on the board are displayed in a window. Clicking on an unselected card selects it; clicking on a selected card unselects it. Clicking on the **Replace** button first checks that the selection is legal; if so, it does the removal and deals cards to fill the empty slots. Clicking on the **Restart** button restarts the game.

The folder **Activity6 Starter Code** contains the file `Elevens.jar` that, when executed, runs a GUI-based implementation. In a Windows environment, you may be able to run it by double-clicking on it. Otherwise you can run it with the command

```
java –jar Elevens.jar
```

Play a few games of Elevens. How many did you win?

Questions:

1. List all possible plays for the board 5♠ 4♥ 2♦ 6♣ A♠ J♥ K♦ 5♣ 2♠

   5+6 (index 0 and 3)
   6+5 (index 3 and 7)

2. If the deck is empty and the board has three cards left, must they be J, Q, and K? Why or why not?

   Yes, because if you draw any other cards, you cannot win (1 remains, and cannot add up to 11)

3. Does the game involve any strategy? That is, when more than one play is possible, does it matter which one is chosen? Briefly explain your answer.

   Somewhat. You can save cards for later (as in make sure future cards can add up to 11), but the cards delt are random. This means you can keep cards to make future pairs, but sometimes that failes to the RNG.
   You can also lose first draw ;(.

Activity 7: Elevens Board Class Design

Introduction:

Now that the `Card` and `Deck` classes are completed, the next class to design is `ElevensBoard`. This class will contain the state (instance variables) and behavior (methods) necessary to play the game of Elevens.

Questions:

1. What items would be necessary if you were playing a game of Elevens at your desk (not on the computer)? List the private instance variables needed for the `ElevensBoard` class.

   Deck deck
   Hand hand

2. Write an algorithm that describes the actions necessary to play the Elevens game.

   for i = deck's size, i down to size - 9
    draw card and place in hand
   end
   while game is running
    if hand's selected point value = 11 or hand's selected = "jack" and "queen" and "queen"
    remove cards from hand
    for i deck's size, i down to size - drawn amount
     draw card and place in hand
    end
    if cards in hand is 0 then
     game is not running
     game is win
    end
    for i = 0, i up to hands size
     for j =0, j up to hands size
      if hand at i and hand at i = 11 then continue

3. Now examine the partially implemented `ElevensBoard.java` file found in the **Activity7 Starter Code** directory. Does the `ElevensBoard` class contain all the state and behavior necessary to play the game?

   Mostly, but some "to impliment" methods need to be implimented, like checking if the game can continue or not. As is, it does not work, but implimented, yes.

4. `ElevensBoard.java` contains three helper methods. These helper methods are `private` because they are only called from the `ElevensBoard` class.

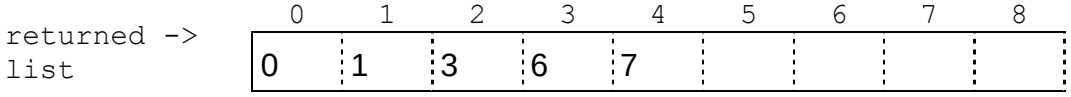
   a. Where is the `dealMyCards` method called in `ElevensBoard`?

When the object is created and when a new game is made.

   b. Which `public` methods should call the `containsPairSum11` and `containsJQK` methods?

anotherPlayIsPossible
isLegal

   c. It's important to understand how the `cardIndexes` method works, and how the list that it returns is used. Suppose that `cards` contains the elements shown below. Trace the execution of the `cardIndexes` method to determine what list will be returned. Complete the diagram below by filling in the elements of the returned list, and by showing how those values index `cards`. Note that the returned list may have less than 9 elements.



d. Complete the following `printCards` method to print all of the elements of cards that are indexed by `cIndexes`.

```
public static printCards(ElevensBoard board) {
   List<Integer> cIndexes = board.cardIndexes();

   /* Your code goes here. */
   for(Integer i : cIndexes) {
   System.out.println(board.cardAt[i]);
   }

}
```

e. Which one of the methods that you identified in question 4b above needs to call the `cardIndexes` method before calling the `containsPairSum11` and `containsJQK` methods? Why?

Is legal. This is because you only need some of the playable cards (the cards at the selected indexes)

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

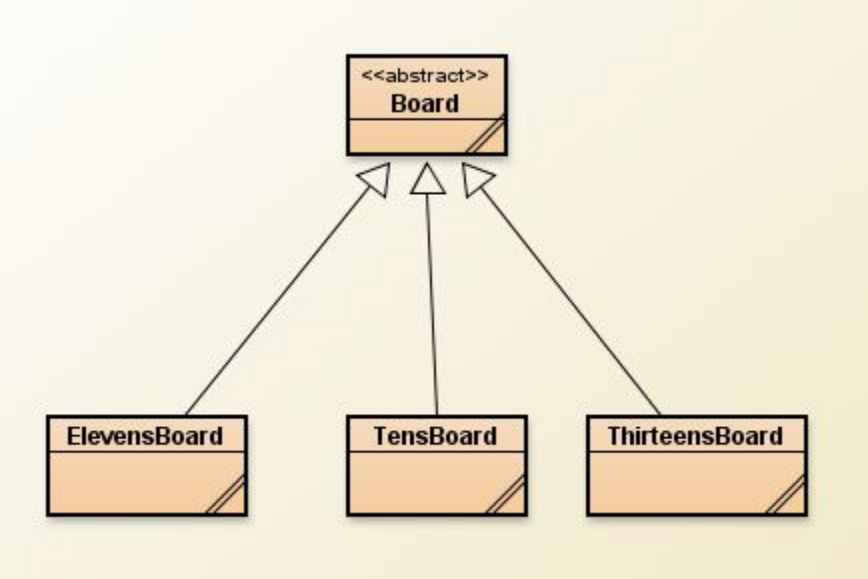
In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- 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*.

   They all need:
   Cards
   decks
   The differences are:
   The numbers that they add up to
   13s takes kings rather then needing KQJ to remove
   10s taking all three QQQ (same for J and K) to remove

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?

   Board b = new ElevensBoard(); //uses the board class, but instanciates it as a elevens board. This will use elevens board stuff, but calls only have to use b.
   b.isLegal() would work because the method is "contained" in the board class, but the Elevens board actually is the thing controlling it.
   polymorphism.

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?

   Yes, because the only things that change are
   anotherPlayIsPossible (deturmine if JKQ, QQQ, or adding up to 10, 11, or 13)
   isLegal (figure out if the move is legal)
   All other methods in elevnsboard are game specific helper methods, and are implimented in the 2 methods above. (so all differences between 10 11 and 13 are covered by the specific class using these 2 methods)

Glossary

assertion: Boolean expressions that should be true if the program is running correctly. The Java `assert` statement can be used to check assertions in a program.

class invariant: A logical statement relating to the values of the instance variables of a class that is always true between calls to the class's methods (also referred to as a "data invariant"). ("Invariant" means "not varying" or "not changing.")

client class: A class that uses another class (e.g., The `Deck` class is a client of the `Card` class.).

helper method: A method, usually `private`, that is called by another method. Helper methods are used to simplify the calling method. They also facilitate code reuse when they provide a function that can be used by more than one calling method.

loop invariant: A logical statement that is always true when execution reaches a loop's termination test.

model: A class with behaviors and state that represent key features of some "real-world" object or process. We say that a class models the "real-world" object. For example, the `Deck` class models a real deck of cards.

perfect shuffle: A card-shuffling method that starts with dividing the deck into two stacks, then interleaving the cards, first a card from stack 1, then a card from stack 2, then another card from stack 1, another from stack 2, and so on.

permutation: A rearrangement of a given sequence of values. There are six permutations of the sequence [1,2,3], namely [1,2,3] (the "identity" permutation), [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1]. If the given sequence contains duplicate values, so will its permutations. For example, the permutations of [1,1,2] are [1,1,2], [1,2,1], and [2,1,1].

polymorphism: A process that Java uses where the method to execute is based on the object executing the method. For example, if `board.anotherPlayIsPossible()` is executed, and `board` references an `ElevensBoard` object, then the `ElevensBoard` `anotherPlayIsPossible` method will be called.

probabilistic: Based on chance or involving the use of randomness.

pseudo-random number generator: A procedure that produces a sequence of values that passes various statistical tests for randomness (e.g., any value is just as likely to occur in a given position in the sequence as any other).

random number generator: See **pseudo-random number generator**.

refactor: Reorganizing code. One example of refactoring is creating helper methods to simplify code or eliminate duplicate code. Another is splitting a class into a superclass and a subclass, putting the code that would be common to other subclasses into the new superclass.

selection shuffle: A card-shuffling method that works similarly to the selection sort. It randomly selects a card for each position in the deck from the remaining unselected cards.

shuffle: A method of permuting (mixing up) the cards in a deck. See **perfect shuffle** and **selection shuffle**.

simulation: Imitation, using a computer program, of some real-world process. The "actors" in the process correspond to objects and variables in the simulation, while the interactions between the actors correspond to program methods.

systematic: Performed using a logical step-by-step process.

truncation: Removal of the fractional part of a real or double value, producing an integer.

References

The Complete Book of Solitaire and Patience Games, by Albert H. Morehead and Geoffrey Mott-Smith, Bantam Books (1977).