# 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 9: Implementing the Elevens Board

### Introduction:

In Activity 8, we *refactored* (reorganized) the original `ElevensBoard` class into a new `Board` class and a much smaller `ElevensBoard` class. The purpose of this change was to allow code reuse in new games such as Tens and Thirteens. Now you will complete the implementation of the methods in the refactored `ElevensBoard` class.

### Exercises:

1. Complete the `ElevensBoard` class in the **Activity9 Starter Code** folder, implementing the following methods.

   **Abstract methods from the `Board` class:**

   a. `isLegal` — This method is described in the method heading and related comments below. The implementation should check the number of cards selected and utilize the `ElevensBoard` helper methods.

```
/**
 * Determines if the selected cards form a valid group for removal.
 * In Elevens, the legal groups are (1) a pair of non-face cards
 * whose values add to 11, and (2) a group of three cards consisting of
 * a jack, a queen, and a king in some order.
 * @param selectedCards the list of the indexes of the selected cards.
 * @return true if the selected cards form a valid group for removal;
 *         false otherwise.
 */
@Override
public boolean isLegal(List<Integer> selectedCards)
```

b. `anotherPlayIsPossible` — This method should also utilize the helper methods. It should be very short.

```
/**
 * Determine if there are any legal plays left on the board.
 * In Elevens, there is a legal play if the board contains
 * (1) a pair of non-face cards whose values add to 11, or (2) a group
 * of three cards consisting of a jack, a queen, and a king in some order.
 * @return true if there is a legal play left on the board;
 *         false otherwise.
 */
@Override
public boolean anotherPlayIsPossible()
```

**`ElevensBoard` helper methods:**

c. `containsPairSum11` — This method determines if the selected elements of `cards` contain a pair of cards whose point values add to 11.

```
/**
 * Check for an 11-pair in the selected cards.
 * @param selectedCards selects a subset of this board.  It is this list
 *                      of indexes into this board that are searched
 *                      to find an 11-pair.
 * @return true if the board entries indexed in selectedCards
 *              contain an 11-pair; false otherwise.
 */
private boolean containsPairSum11(List<Integer> selectedCards)
```

d. `containsJQK` — This method determines if the selected elements of `cards` contains a jack, a queen, and a king in some order.

```
/**
 * Check for a JQK in the selected cards.
 * @param selectedCards selects a subset of this board.  It is this list
 *                      of indexes into this board that are searched
 *                      to find a JQK-triplet.
 * @return true if the board entries indexed in selectedCards
 *              include a jack, a queen, and a king; false otherwise.
 */
private boolean containsJQK(List<Integer> selectedCards)
```

**When you have completed these methods, run the `main` method found in `ElevensGUIRunner.java`. Make sure that the Elevens game works correctly. Note that the cards directory must be in the same directory with your `.class` files.**

### Questions:

1. The size of the board is one of the differences between *Elevens* and *Thirteens*. Why is `size` not an abstract method?

   Assuming you are talking about BOARD_SIZE, that is because all it has to be is a variable. A super constroctor can handle the size of the board, and will always be required by it's sub classes to have.

2. Why are there no abstract methods dealing with the selection of the cards to be removed or replaced in the array `cards`?

   Because you can select cards in all of the games, there is no difference between just selecting cards in 11s, 10s, 13s. Actually replacing the cards is a different story.

3. Another way to create "IS-A" relationships is by implementing interfaces. Suppose that instead of creating an `abstract Board` class, we created the following `Board` interface, and had `ElevensBoard` implement it. Would this new scheme allow the Elevens GUI to call `isLegal` and `anotherPlayIsPossible` polymorphically? Would this alternate design work as well as the `abstract Board` class design? Why or why not?

```
public interface Board
{
   boolean isLegal(List<Integer> selectedCards);

   boolean anotherPlayIsPossible();
}
```

   Yes, you could have different versions of the isLegal method and anotherPlayIsPossible. The interface would NOT work as well because you cannot declare stuff in the interface itself. This would mean you would have to "retype" simmilar things that apply to all of the different games, which can be inconvinent.

# 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).