# Activity 1: Design and Create a `Card` Class

### Introduction:

In this activity, you will complete a `Card` class that will be used to create card objects.

Think about card games you've played. What kinds of information do these games require a card object to "know"? What kinds of operations do these games require a card object to provide?

### Exploration:

Now think about implementing a class to represent a playing card. What instance variables should it have? What methods should it provide? Discuss your ideas for this `Card` class with classmates.

Read the partial implementation of the `Card` class available in the **Activity1 Starter Code** folder. As you read through this class, you will notice the use of the `@Override` annotation before the `toString` method. The Java `@Override` annotation can be used to indicate that a method is intended to override a method in a superclass. In this example, the `Object` class's `toString` method is being overridden in the `Card` class. If the indicated method doesn't override a method, then the Java compiler will give an error message.

Here's a situation where this facility comes in handy. Programmers new to Java often encounter problems matching headings of overridden methods to the superclass's original method heading. For example, in the `Weight` class below, the `tostring` method is intended to be invoked when `toString` is called for a `Weight` object.

```
public class Weight {
   private int pounds;
   private int ounces;
       ...
   public String tostring(String str) {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
       ...
}
```

Unfortunately, this doesn't work; the `tostring` method given above has a different name and a different signature from the `Object` class's `toString` method. The correct version below has the correct name `toString` and no parameter:

```
   public String toString() {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
```

The `@Override` annotation would cause an error message for the first `tostring` version to alert the programmer of the errors.

**Exercises:**

1. Complete the implementation of the provided `Card` class. You will be required to complete:

   a. a constructor that takes two `String` parameters that represent the card's rank and suit, and an `int` parameter that represents the point value of the card;

   b. accessor methods for the card's rank, suit, and point value;

   c. a method to test equality between two card objects; and

   d. the `toString` method to create a `String` that contains the rank, suit, and point value of the card object. The string should be in the following format:

      *rank* `of` *suit* `(point value =` *pointValue*`)`

2. Once you have completed the `Card` class, find the `CardTester.java` file in the **Activity1 Starter Code** folder. Create three `Card` objects and test each method for each `Card` object.

# Activity 2: Initial Design of a `Deck` Class

### Introduction:

Think about a deck of cards. How would you describe a deck of cards? When you play card games, what kinds of operations do these games require a deck to provide?

### Exploration:

Now consider implementing a class to represent a deck of cards. Describe its instance variables and methods, and discuss your design with a classmate.

Read the partial implementation of the `Deck` class available in the **Activity2 Starter Code** folder. This file contains the instance variables, constructor header, and method headers for a `Deck` class general enough to be useful for a variety of card games. Discuss the `Deck` class with your classmates; in particular, make sure you understand the role of each of the parameters to the `Deck` constructor, and of each of the private instance variables in the `Deck` class.

### Exercises:

1. Complete the implementation of the `Deck` class by coding each of the following:

   - `Deck` constructor — This constructor receives three arrays as parameters. The arrays contain the ranks, suits, and point values for each card in the deck. The constructor creates an `ArrayList`, and then creates the specified cards and adds them to the list.
     For example, if `ranks = {"A", "B", "C"}`, `suits = {"Giraffes", "Lions"}`, and `values = {2,1,6}`, the constructor would create the following cards:

     ```
     ["A", "Giraffes", 2], ["B", "Giraffes", 1], ["C", "Giraffes", 6],
     ["A", "Lions", 2], ["B", "Lions", 1], ["C", "Lions", 6]
     ```

     and would add each of them to `cards`. The parameter `size` would then be set to the size of `cards`, which in this example is 6.

     Finally, the constructor should shuffle the deck by calling the `shuffle` method. Note that you will not be implementing the `shuffle` method until Activity 4.

   - `isEmpty` — This method should return `true` when the size of the deck is `0`; `false` otherwise.

   - `size` — This method returns the number of cards in the deck that are left to be dealt.

- `deal` — This method "deals" a card by removing a card from the deck and returning it, if there are any cards in the deck left to be dealt. It returns `null` if the deck is empty. There are several ways of accomplishing this task. Here are two possible algorithms:

  **Algorithm 1:** Because the cards are being held in an `ArrayList`, it would be easy to simply call the `List` method that removes an object at a specified index, and return that object. Removing the object from the end of the list would be more efficient than removing it from the beginning of the list. Note that the use of this algorithm also requires a separate "discard" list to keep track of the dealt cards. This is necessary so that the dealt cards can be reshuffled and dealt again.

  **Algorithm 2:** It would be more efficient to leave the cards in the list. Instead of removing the card, simply decrement the size instance variable and then return the card at `size`. In this algorithm, the `size` instance variable does double duty; it determines which card to "deal" and it also represents how many cards in the deck are left to be dealt. **This is the algorithm that you should implement.**

2. Once you have completed the `Deck` class, find `DeckTester.java` file in the **Activity2 Starter Code** folder. Add code in the `main` method to create three `Deck` objects and test each method for each `Deck` object.

### Questions:

1. Explain in your own words the relationship between a `deck` and a `card`.

2. Consider the deck initialized with the statements below. How many cards does the deck contain?

```
String[] ranks = {"jack", "queen", "king"};
String[] suits = {"blue", "red"};
int[] pointValues = {11, 12, 13};
Deck d = new Deck(ranks, suits, pointValues);
```

3. The game of Twenty-One is played with a deck of 52 cards. Ranks run from ace (highest) down to 2 (lowest). Suits are spades, hearts, diamonds, and clubs as in many other games. A face card has point value 10; an ace has point value 11; point values for 2, …, 10 are 2, …, 10, respectively. Specify the contents of the `ranks`, `suits`, and `pointValues` arrays so that the statement

```
Deck d = new Deck(ranks, suits, pointValues);
```

   initializes a deck for a Twenty-One game.

4. Does the order of elements of the `ranks`, `suits`, and `pointValues` arrays matter?

# Activity 3: Shuffling the Cards in a Deck

### Introduction:

Think about how you shuffle a deck of cards by hand. How well do you think it randomizes the cards in the deck?

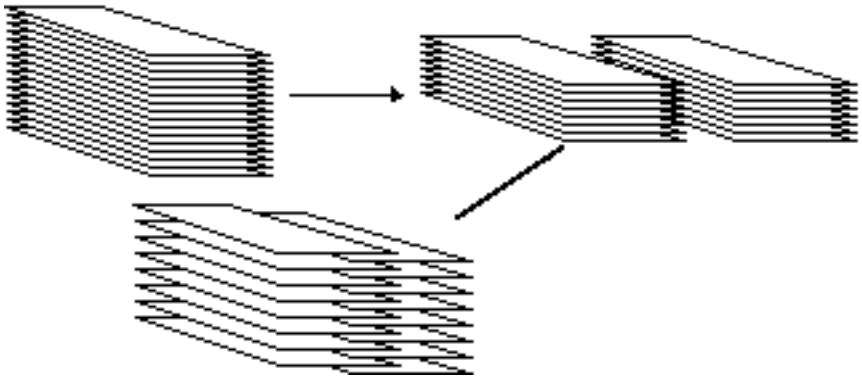
### Exploration:

We now consider the *shuffling* of a deck, that is, the *permutation* of its cards into a random-looking sequence. A requirement of the shuffling procedure is that any particular permutation has just as much chance of occurring as any other. We will be using the `Math.random` method to generate random numbers to produce these permutations.

Several ideas for designing a shuffling method come to mind. We will consider two:

**Perfect Shuffle**

Card players often shuffle by splitting the deck in half and then interleaving the two half-decks, as shown below.



This procedure is called a *perfect shuffle* if the interleaving alternates between the two half-decks. Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight shuffles of a 52-card deck return the deck to its original state!

Consider the following "perfect shuffle" algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`.

> Initialize `shuffled` to contain 52 "empty" elements.
> Set `k` to 0.
> For `j` = 0 to 25,
> - Copy `cards[j]` to `shuffled[k]`;
> - Set `k` to `k+2`.
>
> Set `k` to 1.
> For `j` = 26 to 51,
> - Copy `cards[j]` to `shuffled[k]`;
> - Set `k` to `k+2`.

This approach moves the first half of `cards` to the even index positions of `shuffled`, and it moves the second half of `cards` to the odd index positions of `shuffled`.

The above algorithm shuffles 52 cards. If an odd number of cards is shuffled, the array `shuffled` has one more even-indexed position than odd-indexed positions. Therefore, the first loop must copy one more card than the second loop does. This requires rounding up when calculating the index of the middle of the deck. In other words, in the first loop `j` must go up to `(cards.length + 1) / 2`, exclusive, and in the second loop `j` most begin at `(cards.length + 1) / 2`.

**Selection Shuffle**

Consider the following algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`. We will call this algorithm the "selection shuffle."

> Initialize `shuffled` to contain 52 "empty" elements.
> Then for `k` = 0 to 51,
> - Repeatedly generate a random integer `j` between 0 and 51, inclusive until `cards[j]` contains a card (not marked as empty);
> - Copy `cards[j]` to `shuffled[k]`;
> - Set `cards[j]` to empty.

This approach finds a suitable card for the $k^{th}$ position of the deck. Unsuitable candidates are any cards that have already been placed in the deck.

While this is a more promising approach than the perfect shuffle, its big defect is that it runs too slowly. Every time an empty element is selected, it has to loop again. To determine the last element of `shuffled` requires an average of 52 calls to the random number generator.

A better version, the "efficient selection shuffle," works as follows:

> For `k` = 51 down to 1,
> - Generate a random integer `r` between 0 and `k`, inclusive;
> - Exchange `cards[k]` and `cards[r]`.

This has the same structure as selection sort:

> For `k` = 51 down to 1,
> - Find `r`, the position of the largest value among `cards[0]` through `cards[k]`;
> - Exchange `cards[k]` and `cards[r]`.

The selection shuffle algorithm does not require to a loop to find the largest (or smallest) value to swap, so it works quickly.

### Exercises:

1. Use the file `Shuffler.java`, found in the **Activity3 Starter Code**, to implement the perfect shuffle and the efficient selection shuffle methods as described in the **Exploration** section of this activity. You will be shuffling arrays of integers.

2. `Shuffler.java` also provides a `main` method that calls the shuffling methods. Execute the `main` method and inspect the output to see how well each shuffle method actually randomizes the array elements. You should execute `main` with different values of `SHUFFLE_COUNT` and `VALUE_COUNT`.

### Questions:

1. Write a static method named `flip` that simulates a flip of a weighted coin by returning either `"heads"` or `"tails"` each time it is called. The coin is twice as likely to turn up heads as tails. Thus, `flip` should return `"heads"` about twice as often as it returns `"tails."`

2. Write a static method named `arePermutations` that, given two `int` arrays of the same length but with no duplicate elements, returns `true` if one array is a permutation of the other (i.e., the arrays differ only in how their contents are arranged). Otherwise, it should return `false`.

3. Suppose that the initial contents of the `values` array in `Shuffler.java` are `{1, 2, 3, 4}`. For what sequence of random integers would the efficient selection shuffle change `values` to contain `{4, 3, 2, 1}`?

# Activity 4: Adding a `Shuffle` Method to the `Deck` Class

### Introduction:

You implemented a `Deck` class in Activity 2. This class should be complete except for the `shuffle` method. You also implemented a `DeckTester` class that you used to test your incomplete `Deck` class.

In Activity 3, you implemented methods in the `Shuffler` class, which shuffled integers.

Now you will use what you learned about shuffling in Activity 3 to implement the `Deck shuffle` method.

### Exercises:

1. The file `Deck.java`, found in the **Activity4 Starter Code** folder, is a correct solution from Activity 2. Complete the `Deck` class by implementing the `shuffle` method. Use the efficient selection shuffle algorithm from Activity 3.

   Note that the `Deck` constructor creates the deck and then calls the `shuffle` method. The `shuffle` method also needs to reset the value of `size` to indicate that all of the cards can be dealt again.

2. The `DeckTester.java` file, found in the **Activity4 Starter Code** folder, provides a basic set of `Deck` tests. It is similar to the `DeckTester` class you might have written in Activity 2. Add additional code at the bottom of the `main` method to create a standard deck of 52 cards and test the `shuffle` method. You can use the `Deck toString` method to "see" the cards after every shuffle.

We start by testing the `Card` accessor methods. These tests merely check, using cards with completely different information, that what's stored is what was provided in the constructor. Note the inclusion in the `String` message of information about which value was involved in each assertion.

```
assert c1.rank().equals("ace") : "Wrong rank: " + c1.rank();
assert c1.suit().equals("hearts") : "Wrong suit: " + c1.suit();
assert c1.pointValue() == 1 : "Wrong point value: "
   + c1.pointValue();
assert c6.rank().equals("queen") : "Wrong rank: " + c6.rank();
assert c6.suit().equals("clubs") : "Wrong suit: " + c6.suit();
assert c6.pointValue() == 3: "Wrong point value : "
   + c6.pointValue();
```

Next, we test the `Card` method `matches`. Two cards match if and only if they have the same rank, suit, and point values. A likely implementation of `matches` will involve some comparisons and some uses of `&&`. Common bugs are the copy/paste error mentioned above and the substitution of `||` for `&&`. Comparing `c1` to all the others should reveal these kinds of errors.

```
assert c1.matches(c1) : "Card doesn't match itself: " + c1;
assert c1.matches(c2) : "Duplicate cards aren't equal: " + c1;
assert !c1.matches(c3)
   : "Different cards are equal: " + c1 + ", " + c3;
assert !c1.matches(c4)
   : "Different cards are equal: " + c1 + ", " + c4;
assert !c1.matches(c5)
   : "Different cards are equal: " + c1 + ", " + c5;
assert !c1.matches(c6)
   : "Different cards are equal: " + c1 + ", " + c6;
```

Finally, we test `toString`, again on two completely different objects.

```
assert c1.toString().equals("ace of hearts (point value = 1)")
   : "Wrong toString: " + c1;
assert c6.toString().equals("queen of clubs (point value = 3)")
   : "Wrong toString: " + c6;
```

If all of the tests pass, we provide a message that says so:

```
System.out.println("All tests passed!");
```

**Systematic testing**

Cards didn't involve any data structures more complicated than strings. When testing a class with more complex structures, it makes sense to start small. With a `Deck` class, for example, it might make sense to first provide tests that use a 1-card deck, and then a 2-card deck with different cards.

**Buggy4:**

Constructor or Method (write method name):

Describe a Possible Code Error:

_____

_____

_____

2. Now, examine the Buggy5 folder. This folder contains a `Deck.java` file with multiple errors. Use `DeckTester` to help you find the errors. Correct each error until the `Deck` class has passed all of its tests.

   Note that you may receive a runtime error other than `AssertionError` when running `DeckTester`. If so, you may find it helpful to switch the order of 1-card deck and 2-card deck tests as follows:

```
public static void main(String[] args) {
   test2CardDeck();   // order swapped
   test1CardDeck();   // order swapped
   testShuffle();
   System.out.println("All tests passed!");
}
```

# 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♠

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

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.

# 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

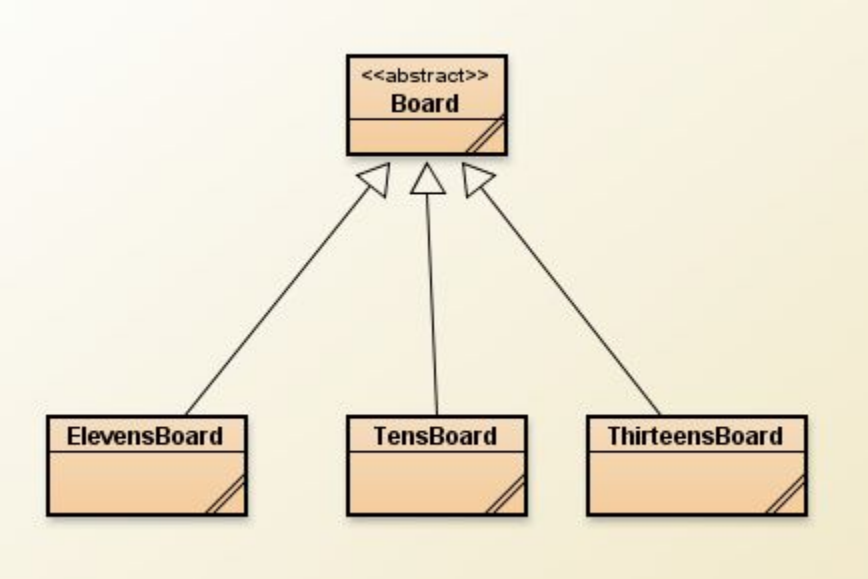
In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- 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?