Activity 2:

1. Cards belong within a deck; a card can function without a deck, but a deck cannot exist without cards.
2. It contains 6 cards.
3. String [] ranks = {“ace”,”king”,”queen”,”jack”,”ten”,”nine”,”eight”,”seven”,”six”,”five”,”four”,”three”,”two”};

String [] suits =

{“hearts”,”clubs”,”spades”,”diamonds”};

int [] pointValues = {11,10,10,10,10,9,8,7,6,5,4,3,2};

1. Yes it does, at least when comparing ranks and point values, since they will always be paired together in the order that it is in.

Activity 3:

1. public static String flip()

{

Random random = new Random();

int val = random.nextInt(3);

if(val == 0 || val ==1)

{

return “heads”;

}

else

{

return “tails”;

}

}

1. import java.util.Arrays;

public static boolean arePermutations(int[] array1, int[] array2)

{

Arrays.sort(array1);

Arrays.sort(array2);

for(int i = 0; i < array1.length(); i++)

{

if(array1[i] != array2[i])

{

return false;

}

}

return true;

}

1. 0, 1, 1 would be the sequence of random integers required to change {1, 2, 3, 4} to {4, 3, 2, 1} with the efficient selection shuffle. The permutations would follow: {1, 2, 3, 4} 🡪 {4,2,3,1} 🡪 {4,3,2,1} 🡪 {4,3,2,1}.

Activity 5:

Buggy 1:

Method with Error: isEmpty()

Possible Code Error: The code checks the incorrect number for size; instead of checking 0, it checks for a value greater than zero.

Buggy 2:

Method with Error: deal()

Possible Code Error: The code subtracts from the instance variable size one too many times.

Buggy 3:

Method with Error: shuffle()

Possible Code Error: The code’s shuffle method is dysfunctional as it does not change the order of the cards in the deck.

Buggy 4:

Method with Error: deal()

Possible Code Error: The deal method does not check if the deck isEmpty; it just returns null.

Buggy 5:

Method with Error: shuffle()

Possible Code Error: The code does not subtract one from the size, thereby resulting in a bounds error with the for loop. Note- I was unable to edit the code in the Buggy 5 file, as it was not there.

Activity 6:

1. All the possible plays include replacing the 5 of spades and 6 of clubs or the 5 of clubs and 6 of clubs.
2. They must be J, Q, and K since the only way one could end up with an odd set of 3 cards at the end of the game is if every number card and ¾ of the face cards were eliminated.
3. The game involves no strategy, since the cards are always dealt out randomly.

Activity 7:

1. The item that would be necessary to play Elevens would be a deck of cards. Private instance variables needed for an ElevensBoard class would be a Deck of cards (a String[] of ranks, a String[] of suits, and an int[] of point values would be needed to make this) and an ArrayList or Array of cards to represent the actual board.
2. Algorithm for Elevens:
   1. Obtain the deck and shuffle it.
   2. Deal out nine cards.
   3. In a loop, deal out two or three cards if the user eliminates two cards or three face cards and check if there are anymore possible moves/combos.
   4. If there are no more moves and cards are left in the deck, the user loses.
   5. If there are no more cards left in the deck, and the user has eliminated everything, the user wins.
3. Although not completely implemented yet, the ElevensBoard does contain all the state and behavior necessary to play the game.
   1. The dealMyCards method is called in the constructor and in the method newGame().
   2. isLegal() and anotherPlayIsPossible() should call the containsPairSum11() and containsJQK() methods.
   3. Returned list selected contains Integer objects:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | -- | 3 | -- | -- | 6 | 7 | -- |

* 1. public static printCards(ElevensBoard board)

{

List<Integer> cIndexes = board.cardIndexes();

for(int i =0; i < cIndexes.size(); i++)

{

if(cIndexes.get(i))

{

System.out.println(board.cardAt(cIndexes.get(i).intValue());

}

}

}

* 1. The anotherPlayIsPossible() method needs to call cardIndexes() first because no ArrayList of cards or Integer values gets passed in to it, so it needs to obtain an ArrayList in order to check if there are any possible moves with the set of cards given.

Activity 8:

1. The differences include the board size, the point values, and the criteria for removing cards (i.e. Thirteens involves removing one king at a time, whereas Tens involves removing four face cards or tens at a time). The similarities are everything else – the deck used, the shuffle method, etc. The similarities are what should be placed in the abstract Board class.
2. Basically, the Board class constructor creates a simple Board that takes in board size, point values, ranks, and suits as parameters. ElevensBoard provides the values for the parameters, and it calls the super or parent class in its constructor, allowing it to use the parent class to create a board for itself.
3. The methods implemented in ElevensBoard cover most of the differences between Elevens, Thirteens, and Tens. Thirteens and Tens also need isLegal and anotherPlayIsPossible methods, but they are implemented slightly differently than in Elevens.

Activity 9:

1. All that the size() method does in Board is return the number of cards on the board, which yields different values for Thirteens and Elevens; therefore, size() applies the same algorithm to both classes and does not have to be abstract.
2. If cards were removed from the array, it would change the board size, as the size() method checks the size of the cards array. Thus, card indexes are removed, and not the cards themselves.
3. Making Board into an interface would not be efficient, as ElevensBoard and ThirteensBoard use many of the methods of the Board class. An interface would not contain code for these methods, so ElevensBoard and ThirteensBoard and any other board classes would have to implement them by themselves. The design could certainly work, but it would not work as well as the abstract Board class design.