

2. (20 points) In this question, I would like you to provide **black box test cases** to determine if a hand in Texas Hold 'Em Poker is a full house or not. For each test case, describe the community and hole cards, whether this hand is a full house or not and why you included this test case. Note that is not necessary for you to provide each of the other types of hands to demonstrate that they are not full houses but it might be worthwhile to include some other specific types of hands if they are similar in some way to a full house. It is not necessary (nor even desirable) for you to write JUnit-like test code.

Here are the relevant parts of the rules from Texas Hold 'Em Poker. (This description is adapted from boardgames.about.com.)

During the course of a game, players are dealt two cards face down, called their hole cards. In addition, five cards are dealt face up in the middle of the table, called the community cards. Players can use any combination of seven cards -- the five community cards and the two hole cards known only to them -- to form the best possible five-card Poker hand. After betting is complete, the player with the best hand wins.

Standard five-card Poker hands are ranked here in order of strength, from the strongest Poker hand to the weakest.

<b>Royal Flush</b>	This is the best possible hand in standard five-card Poker. Ace, King, Queen, Jack and 10, all of the same suit.
<b>Straight Flush</b>	Any five-card sequence in the same suit (e.g.: 8, 9, 10, Jack and Queen of clubs; or 2, 3, 4, 5 and 6 of diamonds).
<b>Four of a Kind</b>	All four cards of the same value (e.g.: 8, 8, 8, 8; or Queen, Queen, Queen, Queen).
<b>Full House</b>	Three of a kind combined with a pair (e.g.: 10, 10, 10 with 6, 6; or King, King, King with 5, 5).
<b>Flush</b>	Any five cards of the same suit, but not in sequence (e.g.: 4, 5, 7, 10 and King of spades).
<b>Straight</b>	Five cards in sequence, but not in the same suit (e.g.: 7 of clubs, 8 of clubs, 9 of diamonds, 10 of spades and Jack of diamonds).
<b>Three of a Kind</b>	Three cards of the same value (e.g.: 3, 3, 3; or Jack, Jack, Jack).
<b>Two Pair</b>	Two separate pairs (e.g.: 2, 2, Queen, Queen).
<b>Pair</b>	Two cards of the same value (e.g.: 7, 7).
<b>High Card</b>	If a Poker hand contains none of the above combinations, it's valued by the highest card in it.

3. (15 points) At the end of this question is a method called `isFullHouse` that could be used in Texas Hold 'Em to determine if a hand contains a full house. If so, it returns `true`. If not, it returns `false`. This method belongs to the `Hand` class.

The `isFullHouse` method uses two instance variables defined in the `Hand` class: `numInRank` and `rankValues`. `numInRank` is set by a `sortCards` method, which must be called before trying to determine the value of the hand. `numInRank` is defined as:

```
/**
 * A count of the number of cards in each rank. Aces appear twice, as
 * LOW_ACE and ACE.
 */
protected int[] numInRank = new int[14];
```

Each entry in the array is a counter. The array is indexed by a card's rank. For example, `numInRank[1]` would indicate the number of 2's in a player's hand. `numInRank[0]` and `numInRank[13]` both indicate the number of Aces, so that an Ace can be considered as being below a 2 or above a King; this may be important for determining straights. `numInRank` accounts for both the player's private cards as well as the community cards that are face-up in the middle of the table and shared by all players.

`rankValues` is defined as:

```
/**
 * The ranks that should be considered for tie-breaking if two hands have
 * the same value. How many rank values are used and what they represent
 * depend on the value of the hand. For example, for a straight, only the
 * first entry in the array is used and it represents the high card in the
 * straight. For two pairs, the first entry is the rank of the high pair,
 * the second is the rank of the second pair, and the third is the rank of
 * the 5th card. -1 signifies that the entry in the array is unused.
 */
protected int[] rankValues = new int[5];
```

For a full house, `rankValues[0]` holds the rank of the three-of-a-kind while `rankValues[1]` holds the rank value of the pair.

Note that it would be correct for `isFullHouse` to return `true` if it contained a full house, even if it also contained a better hand, like a straight flush (if that were possible, anyway!). It would be the responsibility of a different method to check for the types of hands in the order from highest value to lowest value to determine the best way to interpret the player's hand. For example, there might be another method in the `Hand` class like the following:

```

protected void makeBestHand() {
    if (isRoyalFlush ()) {
        value = PokerHandValue.ROYAL_FLUSH;
    }
    else if (isStraightFlush()) {
        value = PokerHandValue.STRAIGHT_FLUSH;
    }
    else if (isFourOfAKind()) {
        value = PokerHandValue.FOUR_OF_A_KIND;
    }
    else if (isFullHouse()) {
        value = PokerHandValue.FULL_HOUSE;
    }
    ...
}

```

Provide thorough **white box testing** for `isFullHouse`. For each test case, describe the input parameters, the expected return value and why you included this test case. This can be done in a tabular format for the incoming parameter values and return value. Your tests should provide good statement, branch and loop coverage with minimal redundancy between tests. That is, each test case should be testing the method in a different way and you should be clear about why you included that test case. It is not necessary (nor even desirable) for you to write JUnit-like test code.

```

/**
 * Returns true if the hand contains a full house. Sets the rank for the
 * highest three-of-a-kind and the highest pair in the hand. sortCards
 * must be called before calling this method so that the numInRank
 * instance variable correctly reflects the cards in the hand.
 * @return true if the hand contains a full house.
 */
protected boolean isFullHouse() {
    boolean tripleFound = false;
    boolean pairFound = false;

    for (int i = numInRank.length-1; i > 0; i--) {
        if (numInRank[i] == 3) {
            rankValues[0] = i;
            tripleFound = true;
            if (pairFound) {
                return true;
            }
        }
        else if (numInRank[i] == 2) {
            rankValues[1] = i;
            pairFound = true;
            if (tripleFound) {
                return true;
            }
        }
    }
    return false;
}

```

4. For this question, I would like you to examine the API for part of the Collections hierarchy provided by Java. The overview comment for `java.util.Collection` states:

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

*Bags* or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. For example, invoking the `addAll(Collection)` method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Reading on we see that `Collection` specifies the following methods as required of all implementations:

- `public int size ()`
- `public boolean isEmpty ()`
- `public Iterator iterator ()`
- `public Object [] toArray ()`
- `public Object [] toArray (Object [] a)`
- `public boolean equals (Object o)`
- `public int hashCode ()`