COMP1006/1406 – Summer 2018

Submit a single file called `assignment3-ID.zip` to cuLearn.
Here ID is your student ID number.

Updated July 24

1: ♠♥♣♦ Cards ♦♣♥♠

A **standard** deck of playing cards consists of 52 cards. Each card has a rank (2, 3, . . . , 9, 10, Jack, Queen, King, or Ace) and a suit (spades ♠, hearts ♥, clubs ♣, or diamonds ♦).

You will create a class called `StandardCard` that will simulate cards from a standard deck of cards. Your class will `extend` the `Card` class (provided).

The ordering of the cards in a standard deck (as defined for this assignment) is first specified by the suit and then by rank if the suits are the same. The suits and ranks are ordered as follows:

suits: The suits will be ordered

$$\text{diamonds} \blacklozenge < \text{clubs} \clubsuit < \text{hearts} \heartsuit < \text{spades} \spadesuit$$

ranks: The ranks will be ordered

$$2 < 3 < \cdots < 9 < 10 < \text{Jack} < \text{Queen} < \text{King} < \text{Ace}$$

A **Joker** card is a special card that is "greater" than any other card in the deck (any two jokers are equal to each other). A joker has no suit "None" from `Card.SUITS`.

Again, the overall ordering for non-joker cards is specified by suit first and then rank; for example, all club cards are "less than" all heart cards. Two cards with the same rank and suit are considered equal.

➡ First

Modify the `Card` class by adding appropriate protected attributes to store the state (suit/rank) of a card.

Next, write a Java class called `StandardCard` that `extends` the provided `Card` class. Your class must have two constructors:

```
public StandardCard(String rank, String suit)
  // purpose: creates a card with given rank and suit
  // preconditions: suit must be a string found in Card.SUITS
  //                rank must be a string found in Card.RANKS
  // Note: If the rank is Card.RANKS[1] then any            <==== update
  //       valid suit from Card.SUITS can be given
  //       but the card's suit will be set to Card.SUITS[4]

public StandardCard(int rank, String suit)
  // purpose: creates a card with the given rank and suit
  // preconditions: suit must be a string found in Card.SUITS
  //                rank is an integer satisfying 1 <= rank <= 14, where
  //                    1 for joker, 2 for 2, 3 for 3, .., 10 for 10
  //                   11 for jack, 12 for queen, 13 for king, 14 for ace
  // Note: as with the other constructor, if a joker is created, any valid suit can be passed
  //       but the card's suit will be set to Card.SUITS[4]
```

Note that the case of strings is important here. The input strings must be exactly the same as those found in `Card.SUITS` or `Card.RANKS`.

The specification for the three `abstract` methods declared in the `Card` class are given by:

```
public int getRank()
  // Purpose: Get the current card's rank as an integer
  // Output: the rank of the card
  //           joker -> 1, 2 -> 2, 3 -> 3, ..., 10 -> 10
  //           jack -> 11, queen -> 12, king -> 13, ace -> 14


public String getRankString()
  // Purpose: Get the current card's rank as a string
  // Returns the cards's rank as one of the strings in Card.RANKS
  //         (whichever corresponds to the card)

public String getSuit()
  // Purpose: Get the current card's suit
  // Returns the card's suit as one of the strings in Card.SUITS
  //         (whichever corresponds to the card)
```

Be sure to add attributes to the abstract `Card` class. You can add any (non-static) attributes and helper methods that you need for your `StandardCard` class.

Example:

```
Card c = new StandardCard("Queen", "Diamonds");
c.getRank();          returns  12
c.getRankString();    returns  "Queen"
c.getSuit();          returns  "Diamonds"
System.out.println(c); displays  12D
Card d = new StandardCard("4", "Spades");
c.compareTo(d);       evaluates to  some negative int
d.compareTo(c);       evaluates to  some positive int
Card e = new StandardCard("Jack", "Spades");
d.compareTo(e);       evaluates to  some negative int
e.compareTo(e);       evaluates to  0
e.getRank();          evaluates to  11
e.getSuit();          evaluates to  "Spades"
Card j = new StandardCard(1, "None");
System.out.println(j); displays  "J"
j.getRankString();    returns  "Joker"
j.getRank();          returns  1
j.getSuit();          returns  "None"
e.compareTo(j);       evaluates to  some negative integer
```

toString() method has been updated to output J for Joker

2: Deck of Cards

Complete the provided `Deck` class. The class has two constructors:

```
public Deck()
  // purpose: creates a deck of 52 standard cards
  //          (one card for each rank/suit combination; no Jokers)

public Deck(int num_jokers)
  // purpose: creates a deck of (52 + num_jokers) cards
  //          consisting of the 52 standard cards
  //          plus num_joker Joker cards
```

You will also complete the following public methods:

```
public List<Card> getCards(int num_cards){return null;}
  // purpose: remove and return num_cards cards from this deck
  //          removes the first num_cards from the deck

public Card getCard(){return null;}
  // purpose: remove and return a single card from this deck
  //          removes the first card from the deck

public void addCard(Card c){}
  // purpose: adds the card c to the back of the deck
```

Update: provided `Deck` class updated to match this specification.

Your must use encapsulation for this class. You can add any private/protected attributes and helper methods that you need. You should not have any static attributes or methods (except possibly a `main` method that you use for testing the class).

Note: You can use any concrete `List` that Java's Collection Framework (JCF) provides.

3: Hand of Cards

Complete the `Hand` class that is provided. You need to implement two methods (`remove` and `add`). You do not need to add very much code to this class.

4: Crazy Eights

Consider the provided abstract `Player` class. You will create a new class called `CrazyEightsPlayer` that is a subclass the `Player` class.

The crazy eights player class simulates a valid player of the game crazy eights. The rules for *our* version of Crazy Eights is as follows:

1. The game starts with each player being given some number of cards from the deck.
2. A single card is taken from the deck and placed on top of the discard pile (and is visible to the player).
3. A player's turn consists of taking zero or more cards from the deck (adding them to their hand) and then playing a card on the top of the discard pile (removing it from their hand).

4. A player can always play a card from their hand that has the same rank as the top card in the discard pile.
5. A player can always play a card from their hand that has the same suit as the top card in the discard pile.
6. A player can always play a card with rank 8. When a player plays an 8, they are allowed to change the suit of the card to any of the four suits of their choosing. (You will do this by removing the eight from your hand and then returning a new Card object with the desired suit.)
7. A player is allowed to (repeatedly) take a card from the deck before playing a card to the discard pile. You must play a card if you are able to. If the deck runs out of cards, the game ends.
8. The player that discards all their cards first is the winner. If the deck is exhausted before any player can play all their cards then there is no winner.

Note: You are NOT implementing a crazy eights game. You are building the classes that would be needed for such a game. You are building the **model** for the game.

Note: Your player should NOT cheat in their play method. When we test your player we will expect no cheating.

Submission Recap

A complete assignment will consist of a single file (`assignment3-ID.zip`) with a single folder in it called `A3`. Inside the A3 folder will the following `.java` files: `Card.java`, `StandardCard.java`, `Deck.java`, `Hand.java` and `CrazyEightsPlayer.java`. You do NOT need to submit the provided abstract classes (which you MUST not change!).