GUI Cards

Please submit one file for each phase (3 .txt files, UML .pdf file)

Understand the Classes and Problem

We wish to move our **Card** classes from the realm of console apps to that of GUI apps. We'll do this in stages.

1. **Read and display Card pictures** - Read **.gif** image files as **Icons**, and attach them to **JLabels** that we can display on a **JFrame**.
2. **Encapsulate the Card Icons in a class GUICard** - Once we debug imagery for cards, above, we can move it into its own class, **GUICard**.
3. **Create a CardTable class** - This **JFrame** class will embody the **JPanels** and **Layout**(s) needed for our application. This is where all the cards and controls will be placed.
4. **Use a CardGameFramework** class - Use an already created class to deal cards for display from an actual deck.
5. **Create the game "High-Card"**

The first phase (item 1) will allow you to debug the problem of reading the **.gif** files and displaying them on a **JFrame** without any excess logic or class complexity. The second phase (items 2 and 3) will let you turn what you did in the first phase into a multi-class project. The final phase (items 4 and 5) will add the **CardGameFramework** class so that your card tools can be combined with your GUI tools to create a GUI program that has real computational power for a GUI card game, "High Card".

The five bullets will be done in ***three phases***. **You should hand in three complete programs, one for each phase.** The main, should be the public class of each program and other classes must *not* be public.

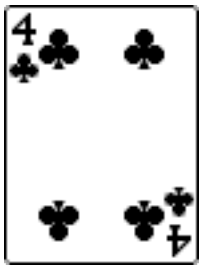
Download the Card .gifs

The GNU Free Software Foundation provides code and images for playing-cards that are public domain. I prepared a **.zip** file which contains a **.gif f**or every card with some minor name changes. Download it here:

Card Images Download.

After you unzip them, you will have a single folder called **images** that contains all the **.gif** files. Move that folder to your **[java workspace]/[project name]** directory. If your project is named **Assignment5**, and your Eclipse workspace is named **workspace**, then move images folder to **workspace/Assignment5**. Since your program considers **Assignment5** to be the root directory, all **.gif** files can be referenced from your program using names like **"images/3S.gif"** for, say, ***3 of spades***.

In addition to the **52 standard cards**, there are **four jokers** and a **card-back** image which can be used to display dealer or other player cards that you don't want the user to see yet.



(The card on the far right is ***joker 2***, not a jack.)

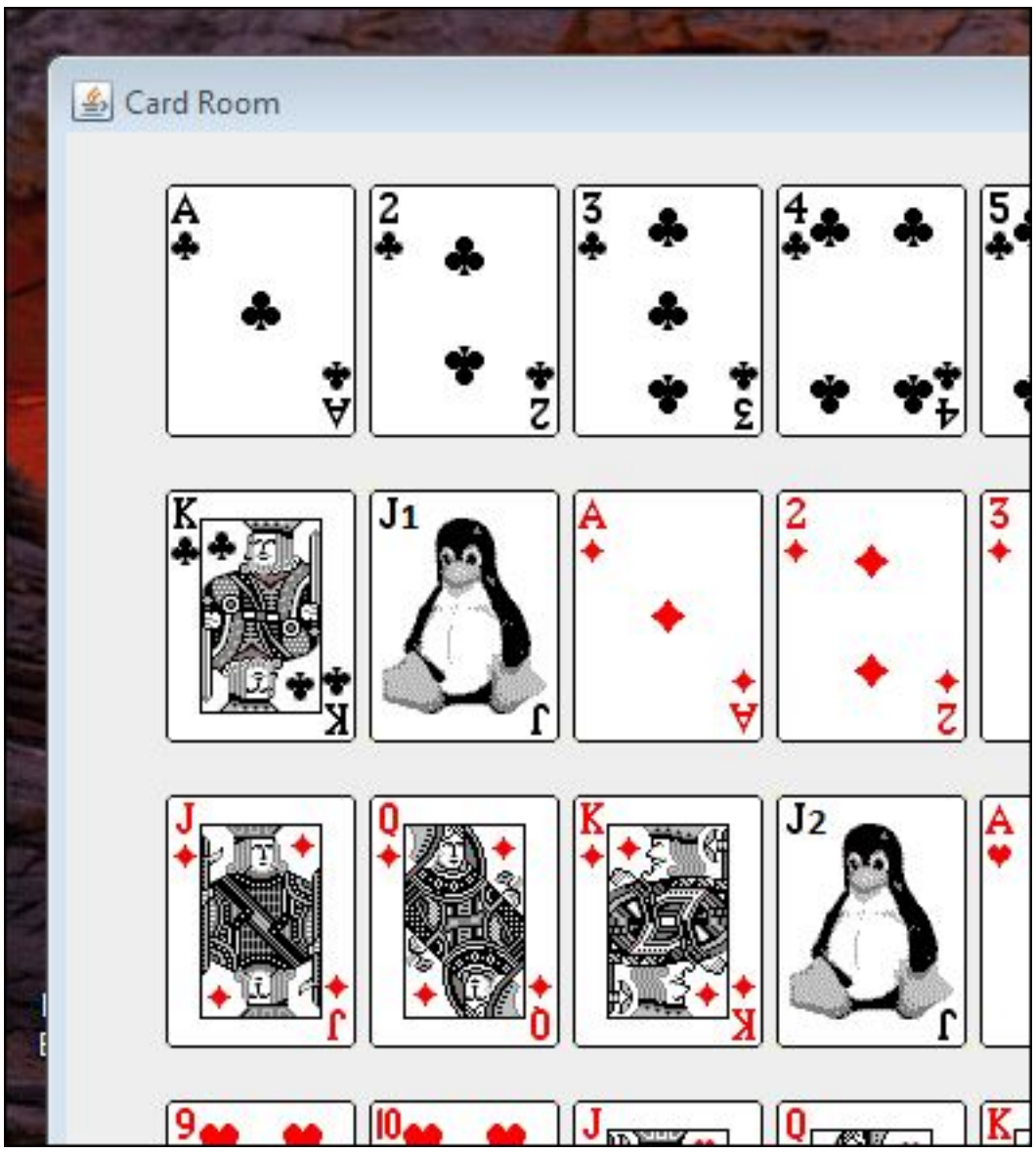
Phase 1: Reading and Displaying the .gif Files

Use what you learned about how to instantiate an **Icon** object to represent any **.gif**, **.jpg** or other image file on your disk and then place that **Icon** on a **JLabel**. In ***Phase 1***, we simply create an array of 57 **JLabels**, attach the **57 .gif files** to them, and display the labels, unstructured, in a single **JFrame**. Here is a possible **main()** that you can use as a starting point. You don't have to use my **main()** but yours should be no longer. Look at your images folder to see the names of each **.gif** file: you have to be able to construct their names in a loop (big point loss if you list all 52 cards named literally because this is array logic). Where is the ***card-back*** image stored? Find it in the **images** folder.

```
import javax.swing.*;
import java.awt.*;

public class Assig5
{
   // static for the 57 icons and their corresponding labels
   // normally we would not have a separate label for each card, but
   // if we want to display all at once using labels, we need to.

   static final int NUM_CARD_IMAGES = 57; // 52 + 4 jokers + 1 back-of-card
```

```
static Icon[] icon = new ImageIcon[NUM_CARD_IMAGES];

static void loadCardIcons()
{
   // build the file names ("AC.gif", "2C.gif", "3C.gif", "TC.gif", etc.
   // in a SHORT loop.  For each file name, read it in and use it to
   // instantiate each of the 57 Icons in the icon[] array.
}

// turns 0 - 13 into "A", "2", "3", ... "Q", "K", "X"
static String turnIntIntoCardValue(int k)
{
   // an idea for a helper method (do it differently if you wish)
}

// turns 0 - 3 into "C", "D", "H", "S"
static String turnIntIntoCardSuit(int j)
{
   // an idea for another helper method (do it differently if you wish)
}

// a simple main to throw all the JLabels out there for the world to see
public static void main(String[] args)
{
   int k;

   // prepare the image icon array
   loadCardIcons();

   // establish main frame in which program will run
   JFrame frmMyWindow = new JFrame("Card Room");
   frmMyWindow.setSize(1150, 650);
   frmMyWindow.setLocationRelativeTo(null);
   frmMyWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   // set up layout which will control placement of buttons, etc.
   FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 5, 20);
   frmMyWindow.setLayout(layout);

   // prepare the image label array
   JLabel[] labels = new JLabel[NUM_CARD_IMAGES];
   for (k = 0; k < NUM_CARD_IMAGES; k++)
      labels[k] = new JLabel(icon[k]);

   // place your 3 controls into frame
```

```
        for (k = 0; k < NUM_CARD_IMAGES; k++)
            frmMyWindow.add(labels[k]);

        // show everything to the user
        frmMyWindow.setVisible(true);
    }
}
```



Hand in a main class that accomplishes this as the first of three programs.

Phase 2: Encapsulating Layount and Icons into CardTable and GUICard Classes

The second part creates a separate **CardTable** class that extends **JFrame**. This class will control the positioning of the panels and cards of the GUI. We also create a new **GUICard** class that manages the reading and building of the card image **Icons**. As a result, some of the machinery and statics that we debugged in the first phase of the main will be moved into one or the other of

these two new classes.

CardTable Class (subclassed from JFrame)

Include five members:

```
static int MAX_CARDS_PER_HAND = 56;
static int MAX_PLAYERS = 2;  // for now, we only allow 2 person games

private int numCardsPerHand;
private int numPlayers;

public JPanel pnlComputerHand, pnlHumanHand, pnlPlayArea;
```

These members are needed is to establish the **grid layout** for the **JPanels**, the organization of which depends on how many cards and players will be displayed. We'll provide an accessor, but no mutator (other than the constructor) for these members. Here are the public instance methods:

- **CardTable(String title, int numCardsPerHand, int numPlayers)** - The constructor filters input, adds any panels to the **JFrame**, and establishes layouts according to the general description below.
- **Accessors** for the two instance members.

Note: We will use three **Public JPanel**s, one for each hand (player-bottom and computer-top) and a middle "playing" **JPanel**. The client (below) will generate the human's cards at random and will be *visible* in the bottom **JPanel**, while the computer's cards will be chosen (again, by the client) to be all ***back-of-card*** images in the top **JPanel**. The middle **JPanel** will display cards that are "played" by the computer and human during the conflict. Let's assume that each player plays one card per round, so for a 2-person game (computer + human) there will be exactly two cards played in the central

region *per round of battle*. My client chose a ***joker*** for the two central cards, just so we would have something to see in the playing region.

GUICard Class

This class is the benefactor of most of the GUI machinery we tested in ***Phase 1***. It will read the image files and store them in a static **Icon** array. Rather than a 1-D array of ***Phase 1***, this will be a 2-D array to facilitate addressing the value and suit of a **Card** in order get its **Icon**. While simple in principle (just read the **Icons** and store them in an array for client use), the details are subtle. We have to be able to convert from **chars** and **suits** to **ints**, and back again, in order to find the **Icon** for any given **Card** object. The overview of the class data and methods, shown below, will suggest the right approach and should take the mystery out of this class.

Include three members:

```
private static Icon[][] iconCards = new ImageIcon[14][4]; // 14 = A thru K
private static Icon iconBack;
static boolean iconsLoaded = false;
```

The 52 + 4 jokers **Icon**s will be read and stored into the **iconCards[][]** array. The ***card-back*** image in the **iconBack** member. None of these data need to be stored more than once, so this is a class without instance data. This class is used is to produce an image icon when the client needs one.

To begin, we need a method that generates the image Icon array from files:

- **static void loadCardIcons()** - the code for this was fundamentally done in ***Phase 1***. The difference here is that we are storing the **Icon**s in a 2-D array. Don't require the client to call this method. Think about where you would need to call it and how can you avoid having the method reload the icons after it has already loaded them once. ***Hint:*** *Call this method any time you might need an* ***Icon****, but make sure that*

it loads the entire array the first time it is called, and does nothing any later time.

The primary public method offered by this class:

- **static public Icon getIcon(Card card)** - This method takes a **Card** object from the client, and returns the **Icon** for that card. It would be used when the client needs to instantiate or change a **JLabel**. It can return something like:

```
return iconCards[valueAsInt(card)][suitAsInt(card)];
```

There is another method that returns the ***card-back*** image:

- **static public Icon getBackCardIcon()** - this one is even simpler than **getIcon()**.

The above three methods comprise the essential part of the **GUICard** class. Everything else is support for these three, so you can work off my implied suggestions, or you can build the class from scratch as you wish. Just make sure you are efficient.

Code From Prior Assignments

We are going to use the classes from Module 3 (Card, Hand, and Deck) but we need to add a couple of things.

Card class

- Adjust for the joker. (Even though there are 4 card icons, think of them as one type, X)

We need a way to know which card is higher when we compare them for the game later. Create the following array and method(s):

- public static char[] valuRanks - put the order of the card values in here with the smallest first, include 'X' for a joker
- static void arraySort(Card[], int arraySize) - will sort the incoming array of cards using a bubble sort routine. You can break this up into smaller methods if it gets over 20 lines or so.

Hand class

Add a sort method:

- void sort() - it will sort the hand by calling the arraySort() method in the Card class.

Deck class

- Adjust for the joker in the MasterPack.

Add methods for adding and removing cards from the deck as well as a sort method. (these will be using in the CardGameFramework given in Phase 3)

- boolean addCard(Card card) - make sure that there are not too many instances of the card in the deck if you add it. Return false if there will be too many. It should put the card on the top of the deck.
- bollean removeCard(Card card) - you are looking to remove a specific card from the deck. Put the current top card into its place. Be sure the card you need is actually still in the deck, if not return false.
- void sort() - put all of the cards in the deck back into the right order according to their values. Is there another method somewhere that already does this that you could refer to?
- int getNumCards() - return the number of cards remaining in the deck.

Main class

You will also need a method that will give you a random new card. It is a **main class method.**

- static Card generateRandomCard() - returns a new random card for the main to use in its tests.

Client for Phase 2

The main class needs to define the specific **JLabel** arrays that will go into each of **CardTable**'s **JPanels**. You will need **NUM_CARDS_PER_HAND JLabels** for the player and the computer (each), even though the computer only uses one **Icon** (***back-of-card***). We also want two **Icon JLabels** for the central **JPanel** (these are the two cards played by computer and human, each turn). But we also need to some **text** below each of the two center icons to we know who played which card (**"Computer"** or **"You"**, so, we'll really need *four* labels in this central play **JPanel** : two for card images and two for text **"Computer"** and **"You"**. Since we want the text directly below the icon, one way to do this is to make your central playing panel a **2x2 Grid Layout**, where the top two positions will be images and the bottom two will be text that describe the images. **Hint**: to center text in a label, use

```
myLabel = new JLabel( "My Text", JLabel.CENTER );
```

The net result should be cards we can see (our hand) in the lower **JPanel** , cards that we can't see -- except for the card backs -- in the upper **JPanel** (the computer's hand) and a central playing region which would represent two cards, one each played by the user and the computer. These two cards depend on what game we are playing, the rules, and the goal. Based on these two cards played, either we or the computer win that round and then we go on to the next round. For this phase, we don't worry about strategy or rules or winning -- we just want to see two cards in the central **JPanel** so we know they are correctly positioned for later program development. Here's a partial picture of a basic solution:

You job is to simply produce this output using the classes and methods suggested. Here is an idea for a **main()** that you can use to get started:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;


public class Assig5
{
   static int NUM_CARDS_PER_HAND = 7;
   static int  NUM_PLAYERS = 2;
   static JLabel[] computerLabels = new JLabel[NUM_CARDS_PER_HAND];
   static JLabel[] humanLabels = new JLabel[NUM_CARDS_PER_HAND];
   static JLabel[] playedCardLabels  = new JLabel[NUM_PLAYERS];
   static JLabel[] playLabelText  = new JLabel[NUM_PLAYERS];

   public static void main(String[] args)
```

```
{
   int k;
   Icon tempIcon;

   // establish main frame in which program will run
   CardTable myCardTable
      = new CardTable("CardTable", NUM_CARDS_PER_HAND, NUM_PLAYERS);
   myCardTable.setSize(800, 600);
   myCardTable.setLocationRelativeTo(null);
   myCardTable.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   // show everything to the user
   myCardTable.setVisible(true);

   // CREATE LABELS -----------------------------------------------
   code goes here ...

   // ADD LABELS TO PANELS ----------------------------------------
   code goes here ...

   // and two random cards in the play region (simulating a computer/hum
   code goes here ...

   // show everything to the user
   myCardTable.setVisible(true);
}
```

Phase 3: Adding CardGameFramework and creating the game "High Card"

We will start by just coping the **CardGameFramework** code from this link, and use it to generate the hands. It makes sure that cards come in and out smoothly in a deck using the methods that you added to the Card, Hand, and Deck classes.

New for the Main:

Start with something like this:

```
numPacksPerDeck = 1;
```

```
numJokersPerPack = 0;
numUnusedCardsPerPack = 0;
unusedCardsPerPack = null;

CardGameFramework highCardGame = new CardGameFramework(
      numPacksPerDeck, numJokersPerPack,
      numUnusedCardsPerPack, unusedCardsPerPack,
      NUM_PLAYERS, NUM_CARDS_PER_HAND);
```

Later, you can use method invocations like

```
...   highCardGame.getHand(1).inspectCard(k)...
```

to access the human player's cards or the computer's cards for the game below. So you instantiate a **CardGameFramework** object, deal the cards, and then read the player's hand, one-card-a-time, producing or updating a **JLabel** for each card.

The source code so far is exactly the same as ***Phase 2***, except:

1. You instantiate a **CardGameFramework** object at the top of **main()**.
2. You **deal()** from it (one statement).
3. In the section where you // CREATE LABELS ..., instead of using **generateRandomCard()** to pick an **Icon**, you use **inspectCard()** to do so.
4. Make sure that it produces the same output as Phase 2 before coding your game below.

5. "High-Card" Game

--You are now perfectly positioned to write a game. You will need an action listener and some rules. The simplest game would be "high-card" in which

you and the computer each play a card, and the high card takes both (which you place somewhere in a winnings[] array, not your hand). You have to add **JLabels** like "You Win" or "Computer Wins". You need to decide how to select a card from your hand like maybe making each card its own button**.** Or other ideas?? Also, how does the computer play? Will you tell it to always try to win by playing the smallest available card that beats yours, and if it can't win, play its smallest card? Maybe it should intentionally lose certain rounds in order to preserve cards. You need to decide who plays first (maybe winner of last round?).

--You need to figure out how to update your cards or the computer's cards to reflect one fewer cards every round so that hands get smaller. This is the fun part!

Phase 4: Create the UML diagram

Draw the UML diagram so that it represents your code structure.

Submission

- No output need be included since it is GUI program.
- Turn in 3 .txt files, one for each code phase.
- UML diagram file.

- **Peer and Self Evaluation**

    - 5% of your grade will be based on your completion of an evaluation of your peers for this assignment and a self evaluation. Put your responses in the box below.
    - Communication -- Them with you? You with them?
    - Who decided what each person would do? Did they do their part? Did you do your part?
    - How much effort did YOU put into the assignment?
    - How hard was this assignment?

- In your own words, describe how your code solved the assignment. This should not be the same as any other team members explanation.