P02 Memory Match Game

Overview

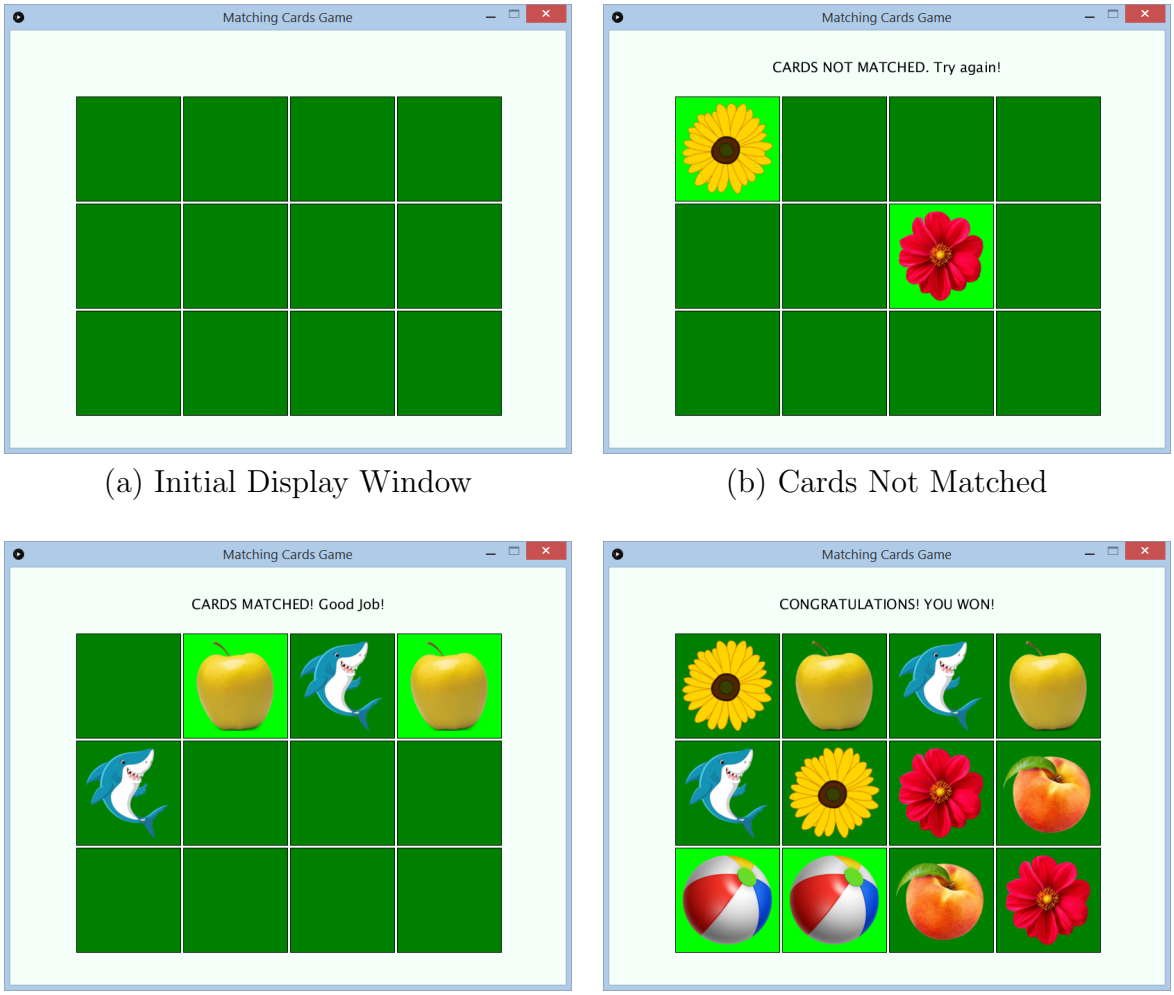
In this assignment, you are going to develop a simple card matching game. We will use the `Processing` library to build the Graphical User Interface and interact with the player. Our game is a one player memory game in which 12 cards are initially laid face down on a surface. Each turn, the player selects two faced down cards, which are then flipped over. If the two cards match, they remain faced up. Otherwise, they are both flipped back to face down. When the player successfully turns all the cards up, the program will declare him a Winner of the game. Fig.1 illustrates an example of what this program might look like when it is done. Besides, a demo video of this game is available here.

Learning Objectives

The goal of this assignment is to practice working with predefined objects (for instance Card objects), and to develop the basis for a simple interactive graphical application. The Graphical User Interface (GUI) application will be written using a provided PApplet object and PImage objects defined within the graphic processing library. This assignment will also give you experience working with callback methods to define how your program responds to mouse-based input or key-pressed.

Grading Rubric

5 points	**Pre-Assignment Quiz:** Generally, you will not have access to this write-up without first completing this pre-assignment quiz through Canvas.
25 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
20 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.



(a) Initial Display Window

(b) Cards Not Matched

(c) Cards Matched

(d) All Pairs of Cards Matched

Figure 1: Matching Game Display Window

1 Additional Assignment Requirements

- The ONLY import statements that you may include in this assignment are:

```
import java.io.File;
import processing.core.PApplet;
import processing.core.PImage;
```

- The automated grading tests in gradescope are **NOT using the full processing library** when grading your code. They only know about the following fields and methods (referenced directly from this assignment). If you are using any other fields, methods, or classes from the processing library, this will cause problems for the automated grading tests. Such references must be replaced with references to one or more of the following:

  - two fields within PImage: int `width`, and int `height`.
  - three fields within PApplet: int `mouseX`, int `mouseY`, and boolean `mousePressed`.
  - three methods within PApplet: PImage `loadImage(String)`, void `image(PImage,int,int)`, and void `main(String)`.

- You MUST NOT add any additional fields either instance or static to your program, and any public methods either static or instance to your program, other than those defined in this write-up.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private static methods to help implement the different public static methods defined in this program, if needed.
- You HAVE TO adhere to the Academic Conduct Expectations and Advice

2 GETTING STARTED

To get started, let's first create a new Java11 project within Eclipse. You can name this project whatever you like, but P02 Matching Game is a descriptive choice. Then, create a new class named `MemoryGame` with a public static void main(String[] args) method stub. This class represents the main class in your program. This process is described near the end of the Eclipse installation instructions here. Do not include a package statement at the top of your `MemoryGame` class (leave it in the default package).

2.1 Download p2core.jar file and add it to your project build path

We have prepared a jar file that contains the processing library, along with a few extra object types to help you build this and future assignments. Download this p2core.jar file and copy it into the project folder that you just created. Then, right-click on this file in the "Package Explorer" within Eclipse, choose "Build Path" and then "Add to Build Path" from the menu. Note: If the .jar file is not immediately visible within Eclipse's Package Explorer, try right-clicking on your project folder and selecting "Refresh".

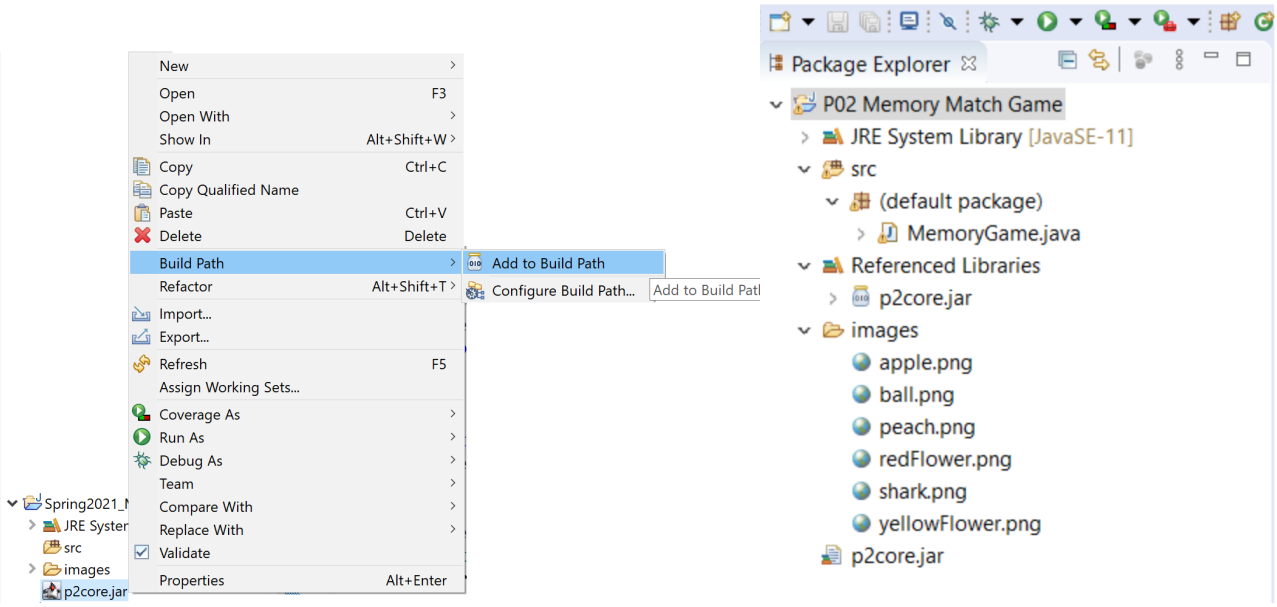
(**Note that** for Chrome users on MAC, Chrome may block the the jar file and incorrectly reports it as a malicious file. To be able to copy the downloaded jar file, Go to "chrome://downloads/" and click on "Show in folder" to open the folder where your jar file is located.)

If the "Build Path" entry is missing when you right click on the jar file in the "Package Explorer", follow the next set of instructions to add the jar file to the build path:

1. Right-click on the project and choose "Properties".

2. Click on the “Java Build Path” option in the left side menu.
3. From the Java Build Path window, click on the “Libraries” Tab.
4. You can add the “p2core.jar” file located in your project folder by clicking “Add JARs...” from the right side menu.
5. Click on “Apply” button.

This operation is illustrated in Fig. 2 (a).



(a) Adding p2core.jar to build path (b) P02 package explorer

Figure 2: Memory Match Game - Getting Started

2.2 Check your project setup

Now, to test that the p2core.jar file library is added appropriately to the build path of your project, try running your program with the following method being called from main() method.

```
Utility.startApplication(); // starts the application
```

If everything is working correctly, you should see a blank window that appears with the title, “Matching Cards Game” as depicted in Fig. 3. `Please consult piazza or one of the consultants, if you have any problems with this setup before proceeding.`

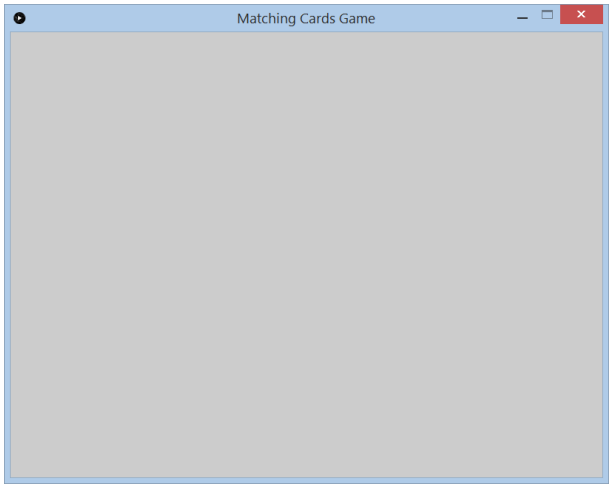


Figure 3: Matching Game - Blank Screen Window

Note that the startApplication() method from the provided Utility class, provided in the p2core jar file, creates the main window for the application, and then repeatedly updates its appearance and checks for user input. It also checks if specific callback methods have been defined in the `MemoryGame` class. Callback methods specify additional computation that should happen when the program begins, the user pressed a key or a mouse button, and every time the display window is repeatedly redrawn to the screen.

2.3 Download images for your Memory Game application

Download the following images.zip compressed folder and extract its content to your project folder. Make sure that the extraction operation results into a folder named images which contains 6 image files with the exact following names: ball.png, redFlower.png, yellowFlower.png, apple.png, peach.png, and shark.png. The organization of your p02 project through eclipse's package explorer is illustrated by Fig. 2 (b).

2.4 Overview of the class Card

The Card class represents the data type for card objects that will be created and used in our `MemoryGame` application. The javadoc documentation for this class is provided here. Make sure to read the description of the constructor and the different methods implemented in the Card class carefully. You do not need to implement this class or any of its declared methods. The class Card is entirely provided for you in p2core.jar. Its implementation details are hidden for you as users of that class. All the information you need to use its public methods are provided in these javadocs Card .

3 VISUALIZE THE GAME DISPLAY WINDOW

3.1 Declare `MemoryGame` Static Fields

Declare final class fields: Let's first declare the different constants (final class fields) that we are going to use in our `MemoryGame` application. Add the following final class fields to your `MemoryGame` class. Make sure to put them outside of any method, including the main(). The top of the class body would be a good placement where to declare them.

```
// Congratulations message
private final static String CONGRA_MSG = "CONGRATULATIONS! YOU WON!";
// Cards not matched message
private final static String NOT_MATCHED = "CARDS NOT MATCHED. Try again!";
// Cards matched message
private final static String MATCHED = "CARDS MATCHED! Good Job!";
// 2D-array which stores cards coordinates on the window display
private final static float[][] CARDS_COORDINATES =
    new float[][] {{170, 170}, {324, 170}, {478, 170}, {632, 170},
                   {170, 324}, {324, 324}, {478, 324}, {632, 324},
                   {170, 478}, {324, 478}, {478, 478}, {632, 478}};
// Array that stores the card images filenames
private final static String[] CARD_IMAGES_NAMES = new String[] {"ball.png", "redFlower.png",
    "yellowFlower.png", "apple.png", "peach.png", "shark.png"};
```

Declare variable class fields: Let's now declare the class variables that we are going to use while developing our game. Add the following static fields to your `MemoryGame` class. Put them outside of any method including, the main(). Putting these static variables just after the final ones would be a good place.

```
private static PApplet processing; // PApplet object that represents
                                   // the graphic display window
private static Card[] cards; // one dimensional array of cards
private static PImage[] images; // array of images of the different cards
private static Card selectedCard1; // First selected card
private static Card selectedCard2; // Second selected card
private static boolean winner; // boolean evaluated true if the game is won,
                               // and false otherwise
private static int matchedCardsCount; // number of cards matched so far
                                      // in one session of the game
private static String message; // Displayed message to the display window
```

Recall that you are allowed to import only the following classes to your `MemoryGame` class: `java.io.File`, `processing.core.PApplet`, and processing.core.PImage.

3.2 Define the setup callback method

Recall that Utility.startApplication() creates the display window, sets its dimension, and checks for callback methods. A callback methods specifies additional computation that should happen when the program begins, the user pressed a key or a mouse button, and every time the display window is repeatedly redrawn to the screen.The first callback method that we will define in this program is the **setup()** method. You should define this method in your `MemoryGame` class with **EXACTLY** the following signature:

```
/**
 * Defines the initial environment properties of this game as the program starts
 */
public static void setup(PApplet processing)
```

Note that setup() method will be run as a result of your call to Utility.startApplication() from main() method.

The **setup()** method receives an argument of type PApplet that is passed to it from the Utility class. It's used to define initial environment properties such as screen size and to load background images and fonts as the program starts. **There can only be one setup() method in the whole program and it is run once when the program starts**.

To convince yourself that this method is being called once when your program starts up, try adding a print statement to the `setup()` method, then run your program. Note that the output of your System.out.println() print statement will be displayed in the console and not on the graphic display window. After this test, you can remove the print statement before proceeding.

3.3 Set the background color of the display window

Notice that we defined a static class variable named `processing` of type PApplet. This variable refers to an object that contains most of the drawing functionality from the processing library. Since this processing object is one that you cannot create at this level on your own, the `Utility.startApplication()` method will pass it as parameter into the setup() method.

To set the background color for the display window using the processing PApplet object, follow the next set of instructions in the setup() method.

1. Set the processing class variable to the one passed as input parameter.
2. Using the processing reference, call the background() method defined in the processing library to set the background color for our Matching Game display window as follows:

```
// Set the color used for the background of the Processing window
processing.background(245, 255, 250); // Mint cream color
```

We have chosen mint cream color, which RGB code is (245, 255, 250) as a background color for our matching game. You can try to change this background color to another color of your choice to see how it works. After that, set the color again to mint cream.

4 DRAW CARDS

4.1 Create and initialize the array images

First, make sure that you have a folder called "images" in your "Project Explorer" and that it contains six images. If it is not the case, go up to subsection 2.3. Recall also that the static field CARD_IMAGES_NAMES is a perfect size array that holds 6 String objects which represent the names of the images stored in the "images" folder. Now, in the setup() method, create the static field images array such that it has exactly the same length as CARD_IMAGES_NAMES array, and stores references of type PImage. Next, let's initialize its content.

Note that the java graphic processing library defines the **PImage** class as a datatype used for storing images. A PImage object is used to represent an image that can be loaded from a file by calling **loadImage()** method, and then drawn to the screen at a given position by calling **image()** method. Processing can display .gif, .jpg, .tga, and .png images.

For instance, the following line of code loads CARD_IMAGES_NAMES[0] (which refers to "ball.png") as a PImage object and stores its reference into images[0].

```
//load ball.png image file as PImage object and store its reference into images[0]
images[0] = processing.loadImage("images" + File.separator + CARD_IMAGES_NAMES[0]);
```

Now, in the **setup()** callback method, load all the image files whose names are stored into CARD_IMAGES_NAMES array and store their references into the images array by calling the method `loadImage()` defined in the processing library.

Draw an image to the display window: You can call the method image() defined in the processing library from the **setup()** method in order to draw one image to the center of the screen. The following line of code, added after initializing the array images, draws a ball to the center of the screen as shown in Fig. 4.

```
// Draw an image of an apple at the center of the screen
processing.image(images[0], processing.width / 2, processing.height / 2);
  // width [resp. height]: System variable of the processing library
  // that stores the width [resp. height] of the display window.
```

Note the importance of adding this code after calling the background() method, instead of before. You can also load and draw another different image (for instance images[1]) to the center of the screen in a similar way. You can also change the position where the image can be drawn to the screen. After that, **remove the lines of code** that draw any image to the screen from your setup() method before proceeding to the next steps. In your setup() method,

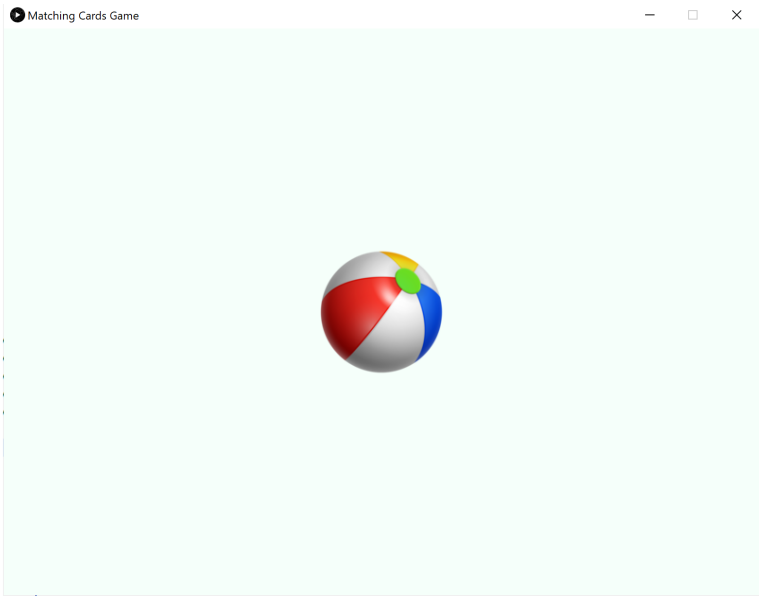


Figure 4: Image of a Ball at the Center of the Screen

leave only the code that sets the processing static field of `MemoryGame` class, sets the color of the background of the display window, creates images array, and initializes its content. When you run your program, you should have an empty mint cream display window.

4.2 Define the startNewGame() method

Let's now initialize the other class variables that we have defined in our `MemoryGame` class. To do so, we are going to define a method called `startNewGame()` with exactly the following signature.

```
/**
 * Initializes the Game
 */
public static void startNewGame(){}
```

Make sure to initialize the following static fields in your `startNewGame()` method with respect to the default values provided in the table below.

static field	initial and restart value
selectedCard1, selectedCard2	null
matchedCardsCount	0
winner	false
message	empty String

The `startNewGame()` method must be called once from the setup() method and each time the

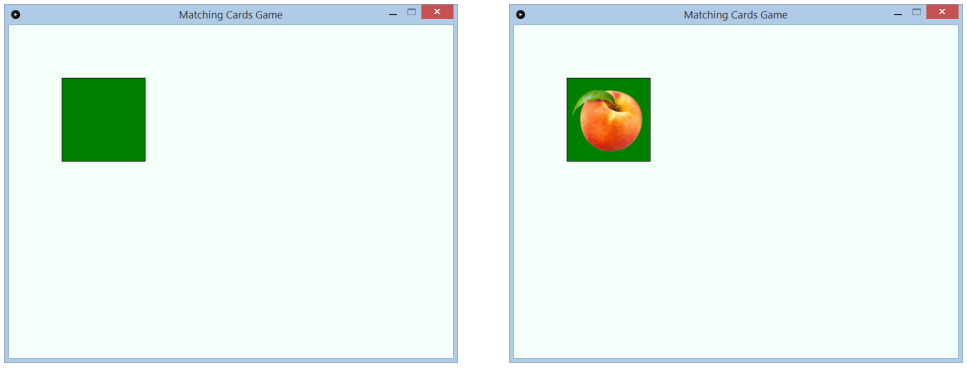
game is restarted. Notice the importance of calling `startNewGame()` method after the lines of code which create and initialize the array images, instead of before.

4.3 Create cards array

The flat surface of our matching game is a 3×4 grid of cards. It can hold up to 12 cards whose coordinates (x,y) are predefined in the static final field CARDS_COORDINATES. In `startNewGame()` method, create the array cards whose length MUST be `CARDS_COORDINATES.length`.

4.4 Draw one card to the display window

Let's now draw one card at a specific position of the game display surface. To do so, in `startNewGame()` method, create an instance of the Card class and stores its reference into a position of your choice of cards array (for instance at index 0). You can set its x and y coordinates to one of the coordinate positions stored in the CARDS_COORDINATES array. Notice that the Card class defines only one constructor, which takes three input parameters: a reference of type PImage that refers to the image of the card, and x, and y of type float representing the coordinates or position where the card will be drawn to the screen. Notice also `isVisible()` and `setVisible()` methods. A card is set visible when it is facing up. A card is not visible (meaning facing down) by default when it is created. The `select()` method selects the card, while `deselect()` method turns off the card selection.



(a) Facing Down Card at Position 0 (b) Facing Up Card at Position 0

Figure 5: One Card Drawn to the Display Window

For instance, the following segment of code added to your `startNewGame()`, draws a card facing down at position 0 of CARDS_COORDINATES array, as illustrated in Fig. 5(a).

```
cards[3] = new Card(images[2], CARDS_COORDINATES[0][0], CARDS_COORDINATES[0][1]);
cards[3].draw();
```

If you set the card to be visible before calling its draw() method, you will have as output the screen shown in Fig. 5(b). Before moving to the next step remove all the lines of code that create any card from your `startNewGame()` method and draw it. cards array should contain only null references.

```
cards[3] = new Card(images[2], CARDS_COORDINATES[0][0], CARDS_COORDINATES[0][1]);
cards[3].setVisible(true);
cards[3].draw();
```

4.5 Create and draw the cards

In this step, we are going to draw 12 cards assigned to random positions within the 3×4 grid display surface. Note that in your implementation, it is highly recommended to avoid use specific values to refer to the length of an array. For instance, you can access the capacity of the `cards` array in your program through the field length defined for Java Arrays.

4.5.1 Mix up the cards

Now, in the `startNewGame()` method, initialize the content of the perfect size array cards such that every PImage reference defined in images array must be assigned to ONLY TWO cards located at different positions selected randomly from the CARDS_COORDINATES array.

The `Utility.shuffleCards(int cardsCount)` method will help you in mixing-up and shuffling the cards randomly at each method call. Passed the number of cards as input, the `Utility.shufflecards()` method returns a perfect size array where the images indexes are shuffled along the cards positions. For instance,

```
Method call: int[] mixedUp = Utility.shufflecards(cards.length);

Output: mixedUp -> {3, 5, 1, 4, 1, 5, 2, 0, 4, 3, 2, 0}
Meaning: The card at index 0 is assigned the image at index 3 of the images array;
         The card at index 1 is assigned the image at index 5 of the images array;
         ..
         The card at index cards.length-1 is assigned the image at index 0 of the images array.
```

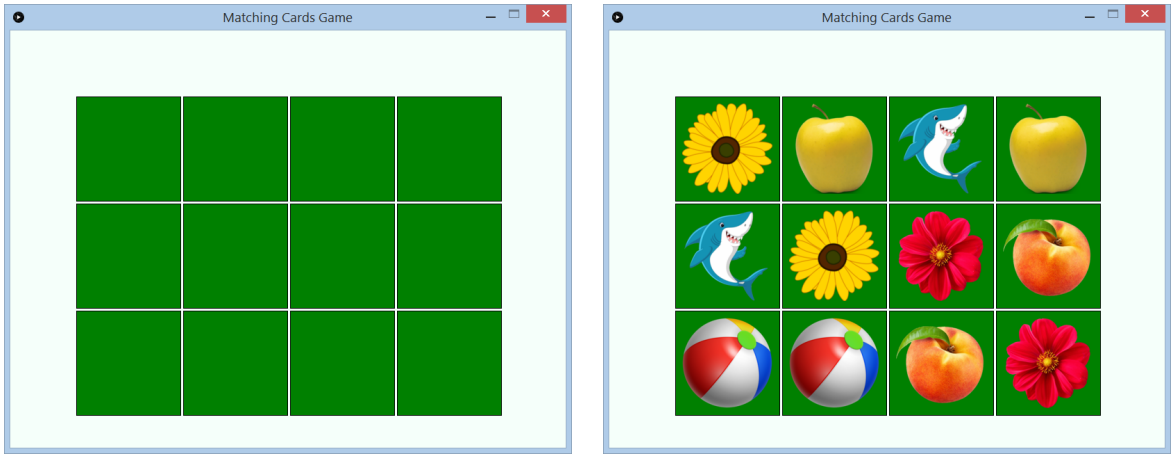
Each time the `Utility.shuffleCards(int cardsCount)` method is called, a new cards distribution will be generated.

4.5.2 Draw the cards

Now, let's draw the mixed up cards to the screen. When created, a card is by default facing down. At this stage, try to set each card visible before drawing it, so you can see the distribution of cards over the grid when drawn. You may have an output similar to the one illustrated in Fig. 6.

5 RESTART THE GAME BY PRESSING N-Key

We would like to initialize the game with a new distribution of the cards each time the key 'N' or 'n' is pressed. To do this, define the callback method keyPressed() with exactly the following signature.



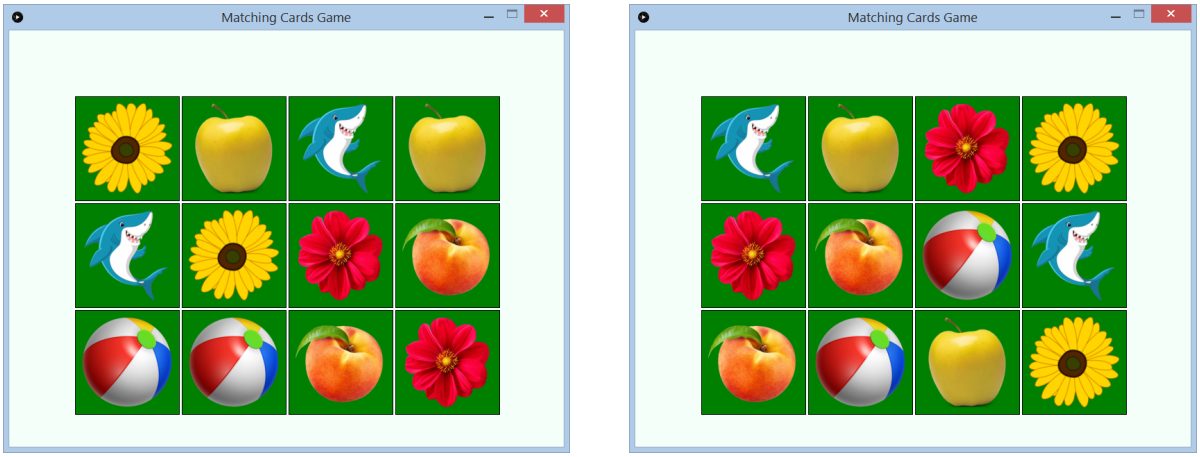
(a) All Cards Facing Down (b) All Cards Facing Up

Figure 6: Grid of Cards

```
/**
 * Callback method called each time the user presses a key
 */
public static void keyPressed()
```

Note that each time the user presses any key, the `keyPressed()` callback method will be called automatically. You can check which key was pressed using the `key` field within the processing object **(processing.key)**.

Now, run your program. You should have an initial distribution of cards. Then, each time you press the N-key, the distribution of cards must change. Fig. 7 illustrates an example of two different distribution of cards.



(a) Initial Distribution of Cards (b) Cards Distribution after N-Key is pressed

Figure 7: Example of Restarting Game

Before moving to the next step, make sure to remove any line of code that sets the visibility of cards to true. When you run your program all cards should be laying in rows, face down.

6 ENABLE SELECTING AND MATCHING CARDS

6.1 Enable selecting cards

Let's now enable the selection of cards. First, we would like to implement the following simple behavior: set a card to be visible and select it when the mouse is pressed and is over the card. We expect so that the display window will be redrawn each time the mouse is pressed and is over a card. To do so, we need to implement another callback method called draw.

6.1.1 Implement the callback draw() method

Add to the `MemoryGame` class a draw() method with exactly the following signature. This method will continuously draw the application display window and updates its content with respect to any change or any event that affects its appearance.

```
/**
 * Callback method draws continuously this application window display
 */
public static void draw()
```

To convince yourself that this method is continuously called by the Utility class as a result of your calling Utility.startApplication(), as long as the application runs, try adding a print statement to the definition of this method. Then, run the program and check the console. After this test, you can remove the print statement. Note also that this method should never be called explicitly in this program.

Now, move the statement that sets the color of the background from setup() method to this draw() method. Move also the code that draws the different cards from `startNewGame()` method to `draw()` method. Finally, call `displayMessage(message)` to draw the class variable `message` to the application display window. We provide you in the following with the implementation details of displayMessage() method.

```
/**
 * Displays a given message to the display window
 * @param message to be displayed to the display window
 */
public static void displayMessage(String message) {
  processing.fill(0);
  processing.textSize(20);
  processing.text(message, processing.width / 2, 50);
  processing.textSize(12);
}
```

6.1.2 Select and turn over a card

In our matching game, initially, all cards laying down on the game surface (meaning not visible and not selected). If the mouse is pressed and is over a card laying down, this card must be turned over and selected. To implement this behavior, let's first implement the `isMouseOver()` method with exactly the following signature.

```
/**
 * Checks whether the mouse is over a given Card
 * @return true if the mouse is over the storage list, false otherwise
 */
public static boolean isMouseOver(Card card)
```

The `isMouseOver()` method should return true if the mouse is over the image of the card object which reference is passed to it as input parameter, and false otherwise. To implement this method, use width and height fields defined within the image of the card to determine whether the mouse is over it. You can access the mouse position through the fields mouseX and mouseY inside the `processing` PApplet class field. `mouseX` and `mouseY` are variables of the processing library that always contain the current horizontal and vertical coordinate of the mouse respectively. To get a reference to the image of the card, call the instance method `getImage()` defined in Card class. As illustrated in Fig. 8, the center of the image is the position (x,y) of the card within the display window.

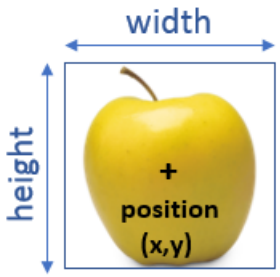


Figure 8: Card Image Dimensions

Now, add the `mousePressed()` callback method to your `MemoryGame` class with exactly the following signature. This method runs each time the mouse is pressed.

```
/**
 * Callback method called each time the user presses the mouse
 */
public static void mousePressed()
```

In the mousePressed() method, check if the mouse is over a card. If it is the case, set the card visible and select it. Note that the Card class defines a method called `select()` that selects the card.

6.2 Matching cards

6.2.1 Implement matchingCards() method

Now, implement the following `matchingCards()` method with exactly the following signature. This method checks whether two given cards match. Note that two cards match if they have the same image.

```
/**
 * Checks whether two cards match or not
 * @param card1 reference to the first card
 * @param card2 reference to the second card
 * @return true if card1 and card2 image references are the same, false otherwise
 */
public static boolean matchingCards(Card card1, Card card2)
```

6.2.2 Update mousePressed() method

Now, using the methods that you have already implemented, implement the in `mousePressed()` method with respect to the following memory match game rules. The `mousePressed` is a callback method which is called automatically each time the user presses the mouse. The following video illustrates how these rules will be implemented.

1. If two cards are selected and they match, keep them visible, and set them to be matched (refer to Card.setMatched() and Card.isMatched() methods). After being matched, the cards must be deselected.
2. If they don't match, turn them back over.
3. The user can select and turn over two cards. If the mouse is over a card which is not already matched, set it to be visible, select it, and update selected cards accordingly.
4. The player wins the game and the `winner` static field is set to true when all the cards have been matched.

You can use the static fields selectedCard1, selectedCard2, matchedCardsCount, and winner to implement the above rules.

It is also worth noting that if a card has been matched, it will remain visible and should not be considered for another selection. Please refer to Card isMatched() and setMatched() methods. If the player is a winner, pressing the mouse won't have any effect.

Feel free to organize this functionality into whatever custom private static methods you see fit. But, make sure that running your program should result in an interaction display window comparable to one shown in Fig. 1.

7 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only ONE file that you must submit is `MemoryGame.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **11:59PM on February 10**th. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.