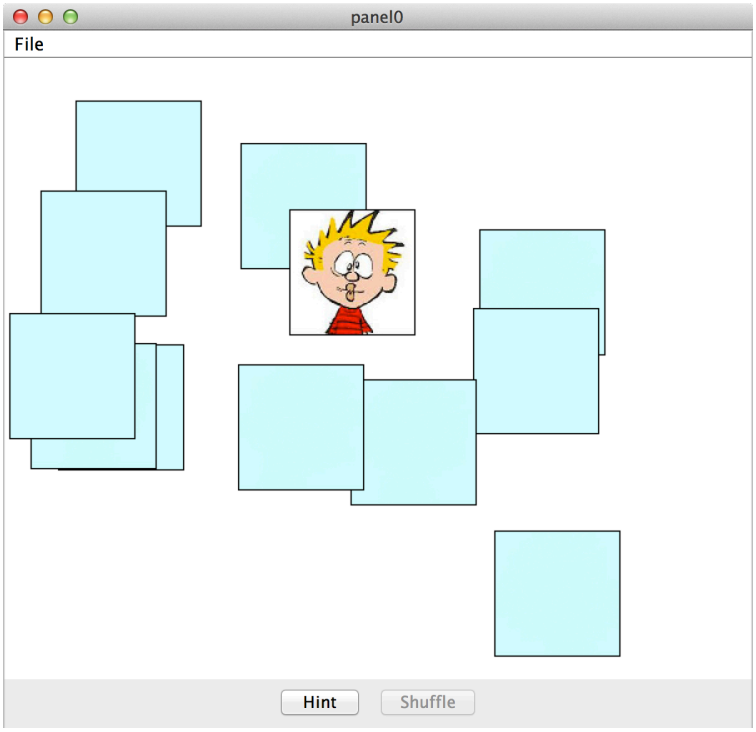
Lab 8

Concentration

Objective To use one-dimensional arrays and implement a collection class.

Your lab assignment this week is to implement the memory game "Concentration." Concentration uses a deck of cards with different pictures on them. All cards are placed face down on a table. The first player flips over two cards. If they have the same image, the player "scores" and takes those two cards off the table. If the two exposed cards have different pictures, they are flipped back over to hide their pictures. The next player then attempts to find a match. The person who can best remember which cards have which pictures will win by making more matches than anyone else.

Our version of Concentration represents the cards as framed squares on the canvas. The program creates a deck of cards with four cards for each image in a set of images. When you click on a card, that card is moved to be on top of all the others, and the image for it appears. The following shows the screen after one card has been clicked on:



If the user clicks on another card with the same image, both images appear and then fade away. When the images are completely obscured again, the two matching cards are removed from the canvas. If a card with a different image is selected, both images appear and then fade away, but the cards remain on the canvas. The game's interface includes two buttons:

- **Hint**: When a single card has been selected, the user may request a "hint" by pressing this button. The program then flips over all other cards with the same image. Those images (other than the one originally selected by the user) then fade back to being hidden. The "Hint" button should only be enabled when one card has been turned over.

- **Shuffle**: When no cards are selected, the user may shuffle the cards using this button. Each card moves to a new, randomly chosen location on the canvas. The "Shuffle" button should only be enabled when no cards have been turned over.

There is a working version of Concentration on the handouts page. You may want to visit the web page as you read the handout so that you can experiment with the program to clarify the details described here.

Design Overview

The design for this lab involves five classes, which we describe in more detail below. We have provided complete implementations for three of these (`Card`, `Fader`, and `Mover`). Your job will be to implement the `CardCollection` class and to complete the implementation of the `WindowController`.

Please bring a design of the CardCollection class and the WindowController's onMouseClick method with you to lab.

- `Concentration`: The `Concentration` class is the `WindowController` for your program. It manages a collection of memory cards and handles button presses and mouse clicks in the canvas. Each memory card is represented as a `Card` object, and the collection of all the cards placed on the canvas is represented as a `CardCollection` object. The `Card` and `CardCollection` classes are described below.

  We have provided some of the code for the window controller. It appears at the end of this handout. You will primarily focus on three parts of the `Concentration` class: setting up the initial collection of cards in `begin`; processing mouse clicks on the canvas in `onMouseClick`; responding to button clicks in `actionPerformed`.

- `Card`: We provide a complete implementation of the `Card` class for you. Here are the constructor and methods available on `Card`s:

```
public Card(double x, double y, int cardSize, String picName,
            Image cardPic, DrawingCanvas canvas)

public Image getImage()
public Location getLocation()
public void move(double dx, double dy)
public void removeFromCanvas()
public void sendToFront()
public boolean contains(Location point)
public String toString()

public void setCoverToClear()
public void setCoverToOpaque()

public void setCoverTransparency(int level)
```

  You create a card by providing the coordinates and size for the card, as well as the picture "printed" on it. The constructor also takes `picName`, the name of the image file for the picture. This parameter is used by `Card`'s `toString()` method.

  Most of the `Card`'s methods are the typical methods for manipulating graphical objects on the canvas. There are also two methods for showing and hiding the card's image: the `setCoverToClear` method exposes the image on a card, and the `setCoverToOpaque` method hides the image. One additional method used for the "fading cover" effect is described below.

- `CardCollection`: A `CardCollection` manages the cards on the screen. You will implement this class. Your class will need to support the constructor and methods listed below:

```
public class CardCollection {

    // Create a new collection with a maximum capacity of maxSize.
    public CardCollection(int maxSize)

    // Add a card to the collection.
    public void add(Card card)

    // Return the first card containing point, starting from the most
    //   recently added card.
    public Card cardAtLocation(Location point)

    // Remove a Card from the collection.
    public void remove(Card card)

    // Return the number of cards currently stored in the collection.
    public int size()

    // Return the card at the given position in the collection.
    // The index parameter must be between 0 and size() - 1.
    public Card cardAtIndex(int index)

    // Return a new collection containing all Cards that
    //   have the given image.
    public CardCollection cardsWithImage(Image image)

    // Return a String representation of the collection suitable
    //   for printing with System.out.println.
    public String toString()
}
```

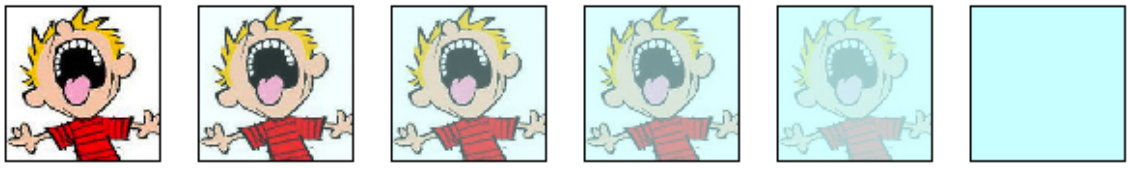
- `Mover` and `Fader`: These two `ActiveObject`s implement the two animations used by the game. We provide implementations that should not need to be modified.

  A `Mover` moves all objects in a `CardCollection` to randomly chosen locations on the canvas. The constructor takes as parameters the collection of cards to move and the maximum x and y coordinates for the new card locations:

  ```
  public Mover(CardCollection cards, double maxX, double maxY)
  ```

  A freshly-created `Mover` object will pick a new location within the given bounds for each card and then move it accordingly. The `Mover` class also supports a boolean `isDone` accessor method to ask whether or not the `Mover` is done moving all of the cards. This method is used by the `Concentration` class to ensure that clicks have no effect while cards are being moved around.

  The `Fader` class is similar in design, but "fades" each card in a collection from having its image visible to not having it visible, as illustrated by following the sequence showing a single card fading:



A `Fader` object performs this animation using one extra method in the `Card` class, namely `setCoverTransparency(level)`. This method takes a parameter, `level`, which is an integer between 0 and 255. That number specifies how transparent the card's cover should be. Setting the transparency to 255 makes the cover be opaque (and the thus image cannot be seen). Setting the transparency to 0 makes the cover completely clear. The sequence above reflects transparency levels of 0, 51, 102, 153, 204, and 255, respectively.

The constructor for the `Fader` class takes two parameters:

```
public Fader(CardCollection cards, boolean removeAfterFade)
```

The first is the collection of cards to fade, and the second indicates whether the cards should be removed from the canvas at the end of the fading.

As with `Mover`s, `Fader`s also support an `isDone()` method so that the window controller can determine whether any animation is currently running.

How to Proceed

First, download a copy of the starter folder from the handouts page and open it in BlueJ. That folder contains the code outlined above, the Calvin images, and some of Steve's favorite cow selfies. You may use any of these images in your programming. But feel free to use your own too!

We suggest that you follow the steps below, which will allow you to proceed in such a way that you can write and test the functionality of each part of your `CardCollection` class and window controller incrementally.

- **Creating the deck of memory cards**. *This step requires adding instance variables and implementing the constructor and* `add` *and* `toString` *methods in* `CardCollection`.

  The first step is to create a deck of cards for the game. In addition to appearing on the canvas, those cards will need to be added to a `CardCollection`. Start by declaring appropriate instance variables in the `CardCollection` class, which will allow you to remember `Card`s in an array. Then implement the constructor for the `CardCollection`, as well as the `add` method.

  The `Concentration` window controller has an array called `imageNames` initialized to contain the names of six of the Calvin images. You can use these images as the initial set for your memory cards. We have also declared an instance variable `cards` to be a `CardCollection`. In the `begin` method, construct the empty `CardCollection`, being sure to specify an appropriate maximum size. Then construct four cards for each image in the array of image names (all at randomly chosen locations on the canvas). After constructing each `Card`, add it to the collection.

  To ensure these steps properly construct the full set of cards, ask each card to show its image by invoking the `setCoverToClear` method on it. Leave the images showing for now.

  To facilitate debugging the later steps, implement a `toString()` method for `CardCollection`, as we did for the collection examples from lecture. Use `System.out.println` to print `cards`. Verify that what is printed matches what is shown on the screen.

- **Selecting Cards.** *This step requires implementing* `CardCollection`*'s* `cardAtLocation` *and* `remove` *methods.*

  Next implement selecting a card and moving it to the front of the canvas. Begin with the `cardAtLocation` method in the `CardCollection` class. This method should find the memory card, if any, that contains the point where the user clicked. If there are multiple overlapping cards that would contain the given point, the top-most card should be returned. So be sure to start searching your collection from the most recently added card, as we did in the drawing program in class. If no card is found to contain the point, the method should return `null`.

You should use the `cardAtLocation` in the window controller's `findSelected` method. This method should find the card in the collection, if any, that contains the click point. It should then bring that card to the front on the canvas. Test it out on one or two cards in different parts of the canvas to be sure you get the desired effect. (You may see the "Hint" and "Shuffle" buttons toggling on and off as you click on images. Don't worry about this — those buttons will behave correctly once you have written more of the `onMouseClick` method.)

Before moving on, remember that the order of the memory cards in the `CardCollection` should reflect the order in which the cards are set out on the canvas. So you will need to make two additional modifications to the `findSelected` method. You will need to remove the selected card from the `CardCollection` and then add it back, as in the drawing program from lecture. Thus, you will need to implement the `remove` method in the `CardCollection` class at this point. [Hint: You might want to write a helper method to find the index of the card.]

Test everything thoroughly before moving on.

- **Selecting two cards.** Now you're ready to handle the selection of a second card. We have already provided some of the necessary code in `onMouseClick`. You will need to add code to remove both selected memory cards if their pictures are the same. Be sure to remove them from the `CardCollection` as well as from the canvas. Do not worry about fading quite yet.

  Once you have thoroughly tested card selection, you're ready to start with images hidden. Modify your `Concentration` class so that the card images are hidden at the start. The images should be shown when memory cards are selected, and if the user picks cards with mismatched images, simply cover back up the images on those cards.

- **Shuffling cards.** *This step requires implementing* `CardCollection`*'s* `size` *and* `cardAtIndex` *methods.*

  Add code to `actionPerformed` to handle the "Shuffle" button. You will create a new `Mover` object to make the shuffling happen. The `Mover` constructor expects three parameters: the collection of cards to shuffle and the maximum x and y coordinates for the new card locations.

  To ensure the window controller ignores additional clicks while the cards are moving, assign your new `Mover` object to the `lastMover` instance variable. The window controller refers to this variable in the `isAnimating()` method.

  Our `Mover` class relies on two methods from your `CardCollection`: `size` and `cardAtIndex`. Before testing the card shuffling you will need to implement them.

- **Fading cards.** Next, add the fading functionality. After a second card is selected, both selected cards should fade. If the cards match, they should disappear after they fade. If they don't match, they should remain on the canvas.

  You will create a `Fader` object to make this happen. The `Fader` constructor expects two parameters, the first of which is a collection of cards to fade. In contrast to how you used the `Mover` above, you don't want to fade the entire collection of memory cards on the canvas – just the two that have been selected. So you'll need to make a new small `CardCollection` with just these two cards in it. The second determines whether or not to remove the fading cards from the canvas at the end of the animation.

  Modify `onMouseClick` to have this behavior when a second card has been selected. As above for shuffling, we have declared an instance variable for a `Fader` called `lastFader` in the window controller for use in `isAnimating`.

- **Giving hints.** *This step requires implementing* `CardCollection`*'s* `cardsWithImage` *method.*

  To add hints, first implement the one remaining unimplemented method in the `CardCollection` class: `cardsWithImage`.

  Then add code to `actionPerformed` in the `WindowController` to provide the hint when the user clicks on the "Hint" button. Once again, you'll make use of a `Fader`. This time, you will

construct a `Fader` and give it the collection of cards found to match the image on the currently selected card – almost. One of the cards in that collection will be the currently selected card itself. This one should not fade away after the hint is provided. So remove it from the collection of "same" cards before you construct the `Fader`.

Extensions

There are many ways to extend this program if you would like to add additional features. Here are a few ideas we had, but be as creative as you like:

- Design your own set of images.
- Keep track of the number of correct and incorrect guesses.
- Enable the user to drag around the cards while playing.
- Provide a "Reset" button to restart the game.
- Provide a combo box for selecting which image set to use, or a slider to adjust how many of each card to use or the card size.
- Add more cards to the screen when the user makes some number of mistakes in a row.
- Pick two or three random cards to shuffle each time the user makes a mistake.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.
- From the "Go" menu at the top of the screen, select "Connect to Server...".
- For the server address, type in "Guest@fuji" and click "Connect".
- A selection box should appear. Select "Courses" and click "Ok".
- You should now see a Finder window with a "cs134" folder. Open this folder .
- You should now see the drop-off folders for the three labs sections. As in the past, drag your "Lab8Concentration" folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click "OK".
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. on Wednesday if you're in the Monday afternoon lab; up to 6 p.m. on Thursday if you're in the Monday night lab; and up to 11 p.m. on Thursday if you're in the Tuesday lab. If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before your lab section's deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like "v2") and the resubmission will work fine.

Starter Code for Concentration WindowController

```
public class Concentration extends WindowController implements ActionListener {

    // Width and height of each Card.
    private static final int CARD_SIZE = 100;

    // Initial window dimensions, and approximate height of
    //   button panel.
    private static final int INITIAL_WIDTH = CARD_SIZE * 6;
    private static final int INITIAL_HEIGHT = CARD_SIZE * 5;
    private static final int BUTTON_HEIGHT = 80;

    // All of the cards on the screen
    private CardCollection cards;

    // The card selected and showing, if any
    private Card selectedCard = null;

    // The last fader created to make Cards appear and fade, and
    //   the last mover created to make Cards move on the screen.
    private Fader lastFader;
    private Mover lastMover;

    // The GUI buttons
    private JButton hintButton;
    private JButton shuffleButton;

    // A constant array of names of all the images to use on the Cards.
    private static final String imageNames[] = {
            "calvin1.png",
            "calvin2.png",
            "calvin3.png",
            "calvin4.png",
            "calvin5.png",
            "calvin6.png"
        };

    /*
     * Set up canvas, buttons, and card collection.
     */
    public void begin() {

        this.resize(INITIAL_WIDTH, INITIAL_HEIGHT + BUTTON_HEIGHT);

        // Create buttons and set up listeners
        hintButton = new JButton("Hint");
        shuffleButton = new JButton("Shuffle");
        hintButton.addActionListener(this);
        shuffleButton.addActionListener(this);

        // Add the buttons to the SOUTH part of the window
        JPanel buttons = new JPanel();
```

```
    buttons.add(hintButton);
    buttons.add(shuffleButton);
    this.getContentPane().add(buttons, BorderLayout.SOUTH);
    this.validate();

    // since there is no selected card, disable the hints button
    hintButton.setEnabled(false);

    cards = new CardCollection(0);  // change me

    // insert code here to initialize contents of collection

}

/*
 *  Return the card in the cards collection at the given point.
 *  That card should become the "top-most" card.
 *  Return null if no cards are at the point.
 */
private Card findSelected(Location point) {
    return null;
}

/*
 * Handle clicks in the canvas.
 */
public void onMouseClick(Location pt) {
    // Ignore clicks if a fader or mover is running
    if (!this.isAnimating()) {

        if (selectedCard == null) {
            selectedCard = this.findSelected(pt);

            // insert code to handle when first card is selected.

        } else {
            Card secondSelected = this.findSelected(pt);

            // is it a valid card and not the same as the first card selected?
            if (secondSelected != null && secondSelected != selectedCard) {

                // insert code to handle when second card is selected

                selectedCard = null;
            }
        }

        // The hint button is enabled only when we have a selectedCard.
        // The shuffle button is the opposite.
        hintButton.setEnabled(selectedCard != null);
        shuffleButton.setEnabled(selectedCard == null);

    }
}
```

```
    /*
     * Handle button presses.
     */
    public void actionPerformed(ActionEvent e) {
        // Ignore events if a fader or mover is running
        if (!this.isAnimating()) {

            // insert code to handle button presses.

        }
    }

    /*
     * Returns true if there is a mover or fader currently animating
     * the Cards on the screen.
     */
    public boolean isAnimating() {
        return (lastMover != null && !lastMover.isDone())
            || (lastFader != null && !lastFader.isDone());
    }

}
```