P05 CABO

Overview

Long ago, four great kingdoms—Cyntra, Avalon, Balthor, and Ophira—warred endlessly, leaving the world in ruins. Their leaders, the Four Crowns, sought peace but could not decide who would lead. The Oracle of the Tower of Whispers spoke: "No more bloodshed. Let a game of magic decide your fate." And thus, the card game CABO was born, each letter honoring the kingdoms.

The game held powerful magic, where low cards wielded strong forces. The lower the rank, the greater its influence. Cards from 7 to 10 could reveal secrets, while those from Jack to Queen could change fortunes in an instant. Most powerful were the Diamond King, a symbol of destiny, who could banish mistakes or seal fate. Around the table, the Four Crowns gathered to play. In CABO, cunning, luck, and skill would decide who would bring peace to the shattered lands.

Now, it is your turn to take up the challenge. As a champion of the kingdom of Cyntra, you will harness a new "magic book"—an enchanted technology we now call Java—to develop and play this legendary game. But your task is not simply to play; it is to create a fully interactive, responsive version of CABO that allows clicks, key presses, and dynamic gameplay.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **10/20**.
+5%	**Bonus Points**: students whose ***final*** submission to Gradescope is before **5:00 PM Central Time** on **WED 10/23** *and* who pass ALL immediate tests will receive an additional 2.5 points toward this assignment, **up to a maximum total of 50 points**.
20 points	**Immediate Automated Tests**: accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
15 points	**Additional Automated Tests**: these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.
10 points	**Manual Grading Feedback**: TAs or graders will manually review your code, focusing on algorithms, use of programming constructs, and style/readability.
50 points	**MAXIMUM TOTAL SCORE**

Learning Objectives

After completing this assignment, you should be able to:

- **Explain** the function of the `@Override` tag and the purpose of *overriding* a method.
- **Implement** a data type in Java which has a parent class other than Object.
- **Utilize** the `processing` library to visually represent an interesting interactive game.
- **Understand** how and when the code in GUI "callback" methods runs.

Additional Assignment Requirements and Notes

Keep in mind:

- Pair programming is **ALLOWED** for this assignment, BUT you must register your partnership ***before the autograder is released***. If you do not do so, you must complete this assignment individually or be subject to Academic Misconduct sanctions.

- The ONLY external libraries you may use in your program are:
  ```
  processing.core.{PImage, PApplet, PConstants},
  java.util.{ArrayList, Collections}, java.io.File
  ```
  Use of *any* other packages (outside of java.lang) is NOT permitted.

- You are allowed to define any **local** variables you may need to implement the methods in this specification (inside methods). You are NOT allowed to define any additional instance or static variables or constants beyond those specified in the write-up.

- You are allowed to define additional **private** helper methods.

- All classes and methods must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.

- Any source code provided in this specification may be included verbatim in your program without attribution.

- All other sources must be cited explicitly in your program comments, in accordance with the Appropriate Academic Conduct guidelines.

- Any use of ChatGPT or other large language models ***must be cited*** AND your submission MUST include **screenshots** of your interactions with the tool ***clearly showing all prompts and responses in full***. Failure to cite or include your logs is considered academic misconduct and will be handled accordingly.

- **Run your program locally before you submit to Gradescope**. If it doesn't work on your computer, *it will not work on Gradescope*.

Need More Help?

Check out the resources available to CS 300 students here:
https://canvas.wisc.edu/courses/427315/pages/resources

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Appropriate Academic Conduct, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - How do I cite my sources?
  - and more!
- Course Style Guide, which addresses such questions as:
  - What should my source code look like?
  - How much should I comment?
  - and more!

Getting Started

1. Create a new project in Eclipse, called something like **P05 CABO**.
   a. Ensure this project uses **Java 17**. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.

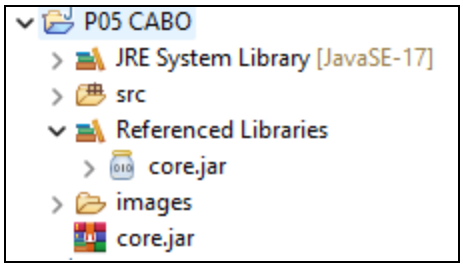
2. Download three (3) Java source file(s) from the assignment page on Canvas:
   a. **Deck.java** (does NOT include a main method)
   b. **AIPlayer.java** (does NOT include a main method)
   c. **CaboGame.java** (includes a main method)

3. Download two (2) more starter files from the assignment page on Canvas:
   a. **core.jar** (the Processing library JAR file, exactly as distributed)
   b. **images.zip** (a zip file containing the images for this game)

4. Create five (5) Java source file(s) within that project's src folder:
   a. **BaseCard.java** (does NOT include a main method)
   b. **ActionCard.java** (does NOT include a main method)
   c. **Hand.java** (does NOT include a main method)
   d. **Player.java** (does NOT include a main method)
   e. **Button.java** (does NOT include a main method)

To complete setup, add **core.jar** to your project as in P02, and add it to your Eclipse build path by right-clicking on it and selecting "Build Path" → "Add to Build Path…"

Unzip the **images.zip** file and add the resulting folder to your project, as a folder of 53 images (52 standard cards and a card back; no jokers).



Verify that your project looks like the one here (note that all of the image files are within an `images` folder, NOT in your src folder). If it doesn't look like this, **STOP** and fix it before you continue.

1. Getting Started

The Javadocs for this program in their entirety are found here: **JAVADOCS**

Read them *CAREFULLY*. Much of what you need to know is contained in them, including which classes to inherit from and which interfaces to implement.

Now that you have the core.jar file in your build path, ***set the provided CaboGame class to inherit from PApplet*** – this is how Processing *really* works, using inheritance rather than static method calls! From now on, consider the CaboGame class to be an object representing the game window.

1.1 BaseCard and its derived classes

Create the **BaseCard** first, and then add its derived class **ActionCard** according to its javadocs.

- If you need help ***creating*** the PImages for the cards (located at
  `"images"+File.separator+rank+"_of_"+suit.toLowerCase()+".png"`
  as, for example, `"images/3_of_hearts.png"`), you can always refer back to P02; we used the same method to create a PImage then, just in a static way.
- ***Drawing*** them is a little different this time: since the provided images are not the same dimensions as WIDTH and HEIGHT, you'll need to use PApplet's method `image(PImage, x, y, width, height)` to get them to display correctly.

1.2 Deck

This class is used to hold the cards for the game – one Deck for the shuffled cards to be dealt, and one Deck for the discard pile. Your provided file contains a static method, `createDeck()`, which will be used to create the cards of various types and shuffle them.

You will need to add the other methods as described in the javadocs, including a `draw()` method so that each pile of cards will show up in the window correctly.

If a deck is currently empty, you should draw an empty rectangle at the given coordinates as follows (this code can – and should – be used verbatim without attribution):

```
// Draw a black rectangle if the discard pile is empty
processing.stroke(0);
processing.fill(0);
processing.rect(x, y, 50, 70, 7);
processing.fill(255);
processing.textSize(12);
processing.textAlign(processing.CENTER, processing.CENTER);
processing.text("Empty", x + 25, y + 35);
```

✅ **CHECKPOINT 1:**
You've finished the basic objects that this game is built around.
Proceed to 1.3 to verify that the Deck is getting set up correctly.

1.3 CaboGame's deckCheck

The **CaboGame** class includes a few basic methods to get you started (and a few less-basic ones to facilitate gameplay with a computer player, that are currently commented out). If you run the class as provided, you should see a window like this, colored green to represent a typical felt card table:

And if you look in the console, you should see the output TODO. This is being generated by the `deckCheck()` method, which is called in the `setup()` callback method.

To test that the card deck is being set up correctly, FIRST add calls to both BaseCard and Deck's `setProcessing()` methods to `setup()` using the current instance of **PApplet**. Calling either constructor without setting the static PApplet variable first will cause an IllegalStateException, if you've followed the javadocs correctly!

[HINT] Think about which of your classes is-a `PApplet`. How do you refer to the current instance of an object while inside that class?

Now complete your `deckCheck()` with these (yes, ***required***) steps:

1. Create a new set of cards with Deck's static method, and store this in a local variable.
2. Verify that this set of cards contains **52 cards**.
3. Verify that it contains **8 of each type** of ActionCard (peek, spy, switch).
4. Verify that it contains **13 of each suit** (Hearts, Diamonds, Clubs, Spades).
5. Verify that if you call `getRank()` on the King of Diamonds, **you get -1 instead of 13**.

This isn't a boolean tester method, so you can just print the results of your tester to the console. If everything is working properly, running your game again might show that same green window and this output in the console (you do NOT need to match this output exactly; `deckCheck()` will be manually graded):

```
Deck size is 52: true
King of Diamonds found!

Found correct numbers of action cards: true
Found correct numbers of each suit: true
```

Since the `setup()` callback method is only called once at the beginning of each program run, you'll only see this output once.

1.4 Add the deck and discard pile to CaboGame

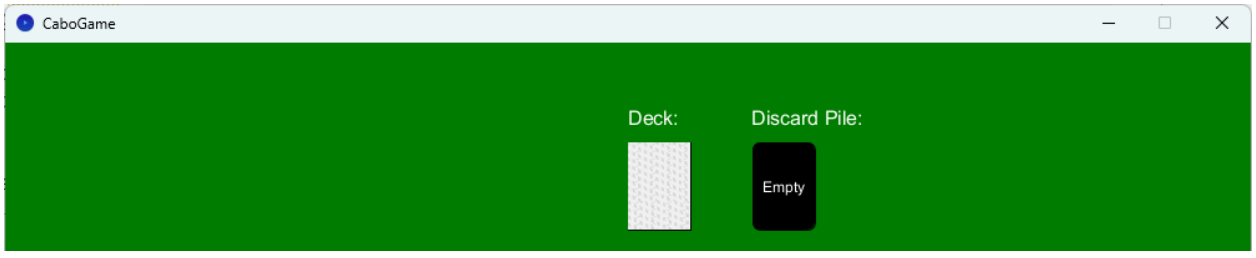
Add the CaboGame data fields `deck`, `discard`, and `drawnCard`, and in the `setup()` method:

1. Set `deck` to be a full, shuffled deck and `discard` to be an empty deck.
2. Initialize `drawnCard` to null.

Add the deck and discard pile to the `draw()` method. The `deck` should be drawn at coordinates (500, 80), and `discard` should be drawn at (600, 80). Add labels to them:

```
textSize(16);
fill(255);
text("Deck:", 520, 60);
text("Discard Pile:", 644, 60);
```

You should see the following results:



It's a little small, but we'll be adding a lot more to the window shortly.

✅ **CHECKPOINT 2:**

Your card decks are showing up, labeled, in the correct places on screen.

2. The CABO Players

Since CABO is a four-player game, we'll be representing each player with their own object in-game. However, since this is not an online game and you're probably not excited to pass a computer around between you and your three closest friends, we've provided an AIPlayer class for you to play against; you just need to code up the base Player class that it inherits from.

2.1 Hand

First, we'll write a mini-Deck for each player to hold - the **Hand** class, which extends Deck. Implement this according to its javadocs; the `HAND_SIZE` should be set to 4 for our purposes.

2.1 Player

Create the **Player** class according to its javadocs, and then add the downloaded **AIPlayer** class to your project (if you already did, it should stop having errors now).

You might notice that this isn't the most *encapsulated* of object-oriented designs; there's no real gameplay encoded into the Player class, and the `getHand()` method just returns the reference to the Hand object directly.

We did this on purpose! The decisions about moves and the interaction between various players' hands – particularly the human player's – rely on interaction with the GUI, which means we need to wait for callback methods to run in order to implement the gameplay. If we're off in the Player class, the GUI can't run, and there's so much interaction between various players in some of the moves of the game that it's actually more straightforward to accomplish this in a slightly more procedural way.

So, y'know, sometimes we do OOP and sometimes we do procedural programming. They're both useful tools. :)

2.2 Back to CaboGame

1. Add private instance variables to CaboGame for the `players` array, and ints indicating the `currentPlayer` (by index into the array) and the `caboPlayer` (the index of the player who has declared CABO).
2. In your `setup()` method (or a private helper method), initialize the `players` array so it can hold four (4) Players, and create a human player named "Cyntra" (the C in CABO) with label 0, and three AI players named "Avalon", "Balthor", and "Ophira" with labels 1, 2, and 3 respectively. Each player's label should correspond to its index in the `players` array.

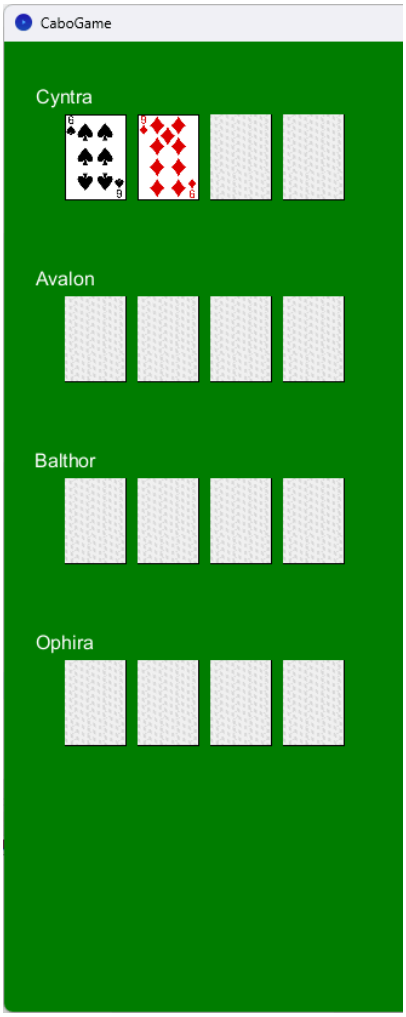
3. Initialize the `currentPlayer` to 0, and set `caboPlayer` to -1.
    a. Using the provided `setGameStatus()` method, add the message "Turn for " + currentPlayer.name to the provided `gameMessages` log. Note the space after "for"!

4. Deal each player 4 cards. If you're being authentic about your dealing, give each player one card and *then* give each player a second card and so on, rather than giving one player four cards and then the next. (This behavior is ***not*** required, but it's a good coding exercise.)

5. Set the human player's first two cards (index 0 and 1) to be face-up for the start of the game. (Recall you can access any player's hand directly using the `getHand()` accessor method.)

✅ **CHECKPOINT 3:**
There are four players who each have four cards, and the deck has 36 cards remaining.
Add print statements to your `setup()` method to verify.

2.3 Draw the Game



Time to focus on the CaboGame's `draw()` method. There's a lot here, so you might want to make one or more private helper methods to organize this code.

1. Write each player's name in the area of the window that will represent their hand. You'll need to set the textSize to 16 and the fill to 255 (white) as you did with the deck and discard pile labels, and then write each player's name at
   `(50, 45 + 150*playerIndex)`

   Run your code. You should see the four players' *names* spaced down the side of the window as in the image on the right, but no cards yet.

2. Now call the `draw()` method for each player's hand. The y-coordinates to use to get these nicely spaced are:
   `(60 + 150*playerIndex)`

   Run your code. You should see the cards appearing as in the image on the right, with the first two cards under Cyntra visible and everything else showing the back of the card.

The cards are dealt and it's now your turn! But you can't interact with the game yet. Time to make some buttons.

3. Buttons

To indicate how you want to take your turn, we'll use a combination of clicking on cards and clicking on buttons that indicate the actions you'd like to take. To help players who might not know the exact mechanics of the game, buttons corresponding to actions which are NOT currently legal will not be clickable.

3.1 The Button class

Create the **Button** class according to its javadocs. To `draw()` a Button:

- **Active** buttons have `fill(200)` *unless the mouse is over them*, in which case they have `fill(150)`; **inactive** buttons have `fill(255, 51, 51)`.

  (These represent two different approaches to communicating color – the one-argument `fill()` uses the same number for all three RGB values, and ends up with a grey color, while the three-argument `fill()` specifies a level for red, green, and blue respectively.)

- To draw a button, use PApplet's `rect()` method again to draw a rectangle at this button's (x,y) position with its specified width and height, and use a fifth parameter value of 5 to make the corners rounded.

- Change the `fill` to 0 (black) and the `textSize` to 14, and align the text using PApplet's `textAlign()` method and `PApplet.CENTER` as BOTH arguments (to align the text both vertically and horizontally).

- Draw the button's label using PApplet's `text()` method and the coordinates of the center of the button. Remember that for this project, (x,y) means the top left corner!

3.2 Add buttons to CaboGame

Add a Button array called `buttons` with length 5 to your CaboGame class and initialize it in the `setup()` method or a private helper as follows:

```
buttons[0] = new Button("Draw from Deck", 50, 700, 150, 40);
buttons[1] = new Button("Swap a Card", 220, 700, 150, 40);
buttons[2] = new Button("Declare Cabo", 390, 700, 150, 40);
buttons[3] = new Button("Use Action", 390 + 170, 700, 150, 40);
buttons[4] = new Button("End Turn", 390 + 170 + 170, 700, 150, 40);
```

You'll need to `setProcessing()` for the Button class at the beginning of `setup()`, too.

3.3 Update Button states with game logic

Complete the `updateButtonStates()` method in CaboGame. This is your first taste of game logic!

- If a computer player is the active player, all buttons should be set to **inactive**.
- Otherwise if a card has not yet been drawn, the only buttons that may be active are to draw a card or declare CABO – and CABO may only be declared if no one else has done so first (that is, the `caboPlayer` still has the -1 you gave it in `setup()`).
- *Otherwise*, the buttons to Draw from Deck and Declare CABO will be **inactive**, and the Swap a Card and End Turn buttons will be **active**.

  If you've drawn an ActionCard you will also be able to use the Use Action button – additionally, ***update the label*** of the Use Action button to be the actionType of that card! (Hard-coding button indexes is permitted.)

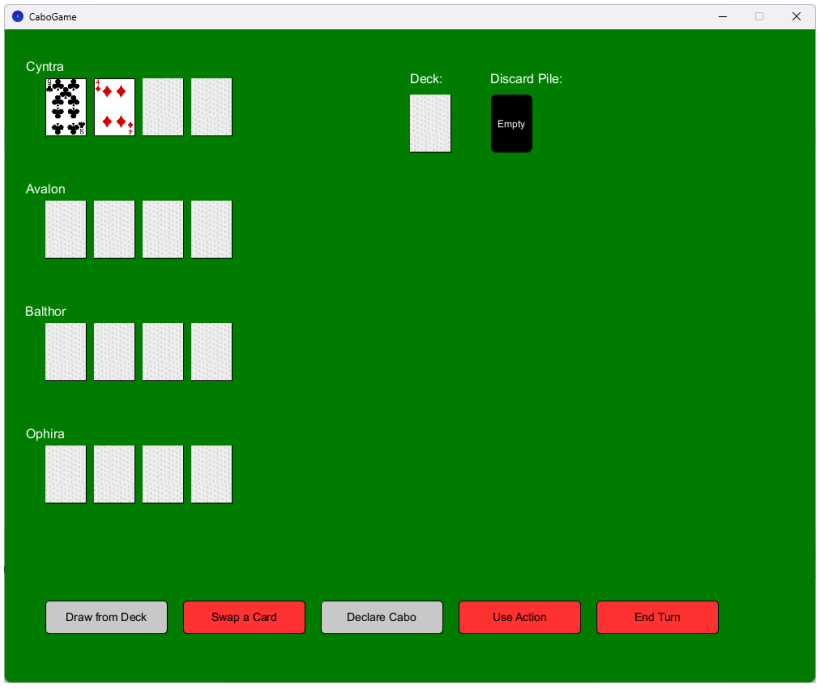
Add a call to `updateButtonStates()` at the end of your `setup()` method, and **run your program**.

…Nothing new! We still need to actually draw these.

3.4 Draw the buttons

You've already done the hard work on this part; all you need to do is add calls to each button's `draw()` instance method from CaboGame's `draw()` method.

(If a computer is playing, don't draw the buttons. It doesn't use them.)



✅ **CHECKPOINT 4:**

You have 5 buttons, 3 of which are red (inactive). Unfortunately none of them actually do anything yet (besides the active ones, which should darken slightly when you hover the mouse over them).

3.5 CaboGame's mousePressed and helper methods

Move to the `mousePressed()` method in CaboGame, which relies on a number of helper methods. We've provided detailed TODOs for these helper methods in the code skeleton.

You should also add the rest of the CaboGame data fields now, if you haven't done so already.

In `mousePressed()`, if a button is ACTIVE and also being clicked on (the mouse is over it):

- **Draw from Deck**: use the `drawFromDeck()` method.
- **Swap a Card**: set the game's `actionState` to SWAPPING (see the provided enum) and update the gameMessages log (see the provided `setGameStatus()` method) to read "Click a card in your hand to swap it with the drawn card." You'll handle the actual swapping later.
- **Declare Cabo**: use the `declareCabo()` method. This will also end the current player's turn (see End Turn below).
- **Use Action**: reset the label of this button to Use Action before you continue, and make sure you add the `selectedCardFromCurrentPlayer` instance variable before you start handling the switch action, and initialize it to -1 in the `setup()` method!

  Get the `ACTION_TYPE` from the drawn card (which will be an ActionCard if this button is active) and set the game's `actionState` accordingly. Give the user some directions for how to proceed using the gameMessages log:

  - **peek**: update the log with "Click a card in your hand to peek at it."
  - **spy**: update the log with "Click a card in another player's hand to spy on it."
  - **switch**: update the log with "Click a card from your hand, then a card from another Kingdom's hand to switch."
- **End Turn**: use the `nextTurn()` method.

What's left is to handle the action-specific helper methods, like `handleCardSwap()`. The starter code has detailed TODOs for you, so finish these up before you continue.

Now that you can draw a card, make sure that you have added code to show it in the `draw()` method. The `drawnCard` should be **face-up** and drawn at (500, 500) when it exists.

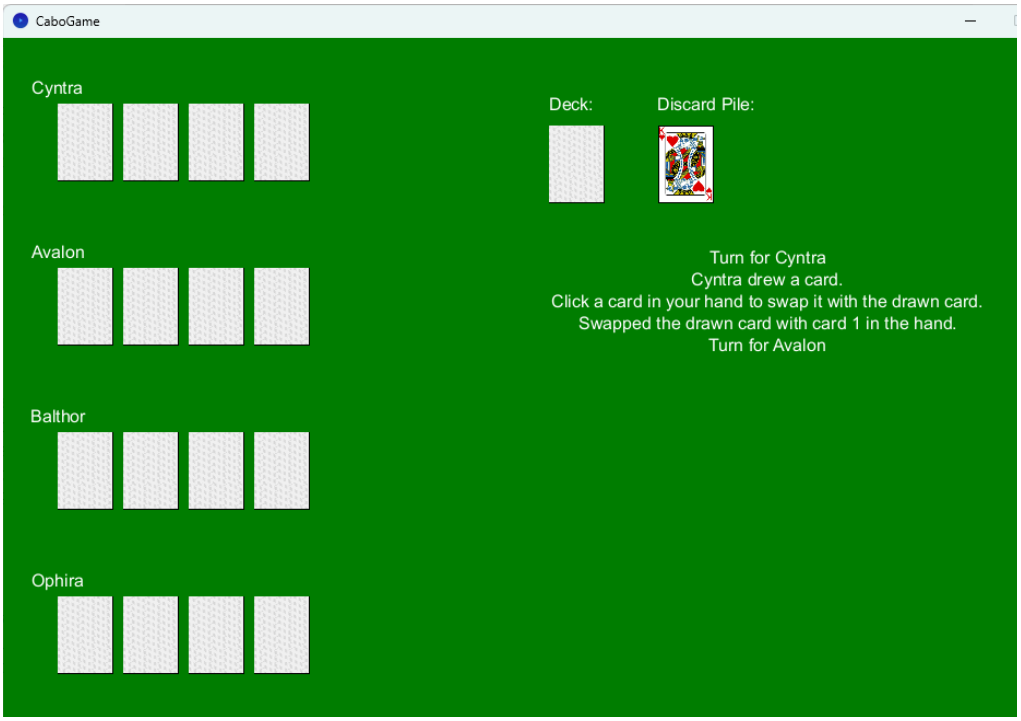
3.6 Show the game message log

To show the game message log (that you've started populating in the previous section), add the following code to your `draw()` method:

```
// Display game messages with different colors based on the content
int y = 200; // Starting y-position for messages
for (String message : gameMessages) {
  textSize(16);
  if (message.contains("CABO")) {
    fill(255, 128, 0);
  } else if (message.contains("switched")) {
    fill(255, 204, 153);
  } else if (message.contains("spied")) {
    fill(255, 229, 204);
  } else {
    fill(255);
  }
  text(message, width - 300, y); // Adjust x-position as needed
  y += 20; // Spacing between messages
}
```

This will show the most recent 15 game messages with some color-coding for some of the actions.

Now you should be able to run the game, take actions (like draw a card), and get to the end of *your* turn:



You're almost done!! Now we just need to allow the computer players to take their turns.

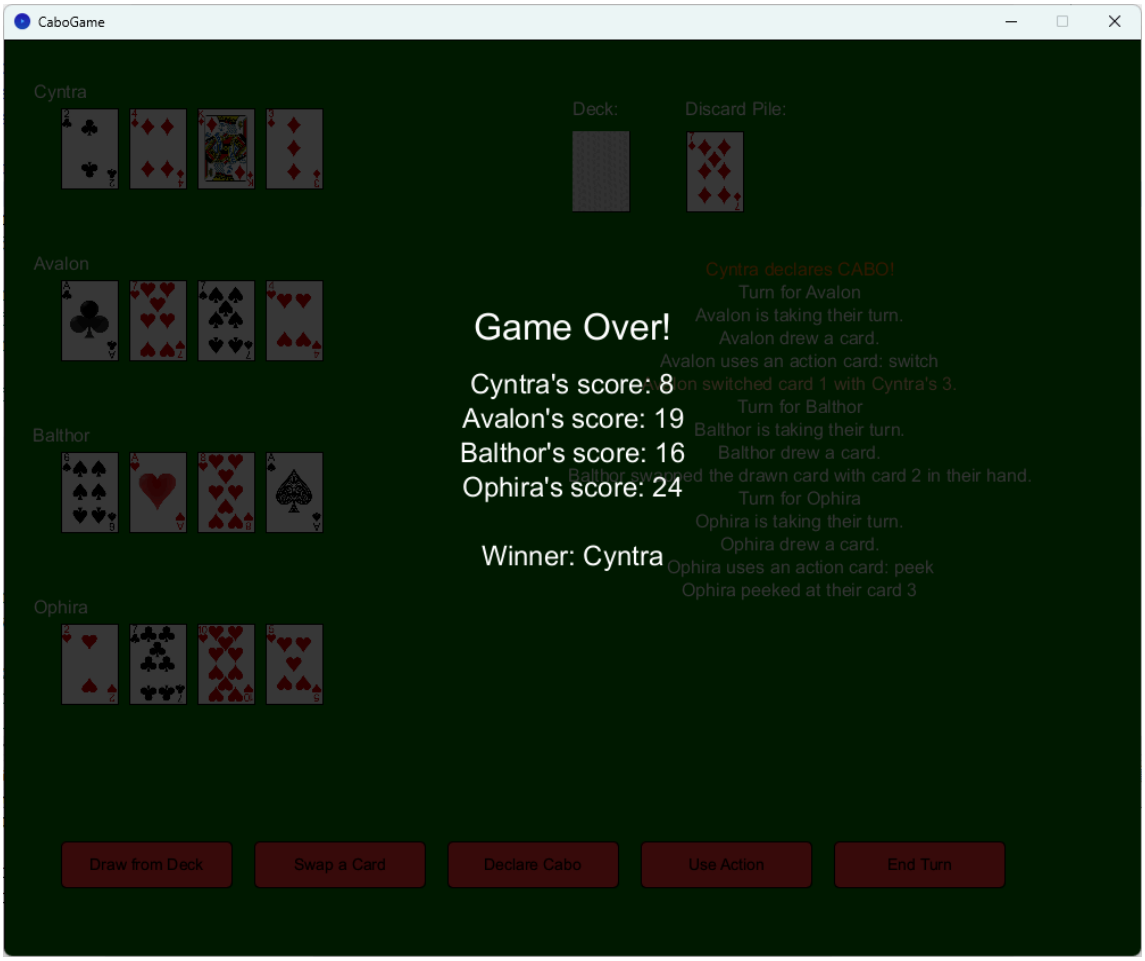
✅ **CHECKPOINT 5:**
You can complete an entire turn as a human player.
Run the game multiple times, verifying that ALL possible actions work correctly.

4. Final Gameplay

The last things you'll need to do are:

- uncomment the two provided AI action methods at the bottom of the CaboGame file,
- add a call in the `draw()` method to `performAITurn()` if the current player is a computer (and the game is not yet over),
- and complete the `displayGameOver()` method.

Once you've done this, you can run an ENTIRE game of CABO with multiple turns (you'll notice the computer players' turns go very quickly, but their actions are listed in the log). Try out the game and see if you can beat our AI!!



Yes, you're reading the game log right: a computer player gave me the King of Diamonds AFTER I declared CABO. I'm just smart and pretty and deserve nice things like that.

Assignment Submission

Hooray, you've finished this CS 300 programming assignment!

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, make a final submission of your source code to Gradescope.

For full credit, please submit the following files (**source code**, *not* .class files):

- `ActionCard.java`
- `BaseCard.java`
- `Button.java`
- `CaboGame.java`
- `Deck.java`
- `Hand.java`
- `Player.java`

Additionally, if you used generative AI at any point during your development, ***you must include screenshots*** showing your FULL interaction with the tool(s).

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Students whose final submission (which must pass ALL immediate tests) is made before 5pm on the Wednesday before the due date will receive an additional 5% bonus toward this assignment. Submissions made after this time are NOT eligible for this bonus, but you may continue to make submissions until 10:00PM Central Time on the due date with no penalty.

Copyright notice