CMPT 201 Project - Uno

Steps 1–2 due April 7 - commit whatever you have to GitHub

Code review meeting between April 8 - April 12th

Steps 3–5 due Sunday, April 14th

Steps 6–7 due Wednesday, April 24th

Project Notes due Sunday, April 28th

In this project you will be completing an implementation of the popular card game Uno. You will also write some *players*, Java classes that represent one person playing the game and their strategy. The student who writes the winning strategy will earn 105% on the final project (full credit plus extra credit)!

The Official Rules

Acquaint yourself with the rules of Uno, in particular what kinds of cards there are and what effects they have. Since you will be writing Uno players that will compete against those of other students, it's important that we're all playing by the same rules! Briefly, Uno is a popular card game in which each player holds a hand of cards, and tries to be the first one to "go out," or play all of their cards. Players

are seated in a ring, and in the middle of this ring is an "up card" or "discard," a card placed on the table face up. Players take turns in sequence, clockwise around the ring. When it is your turn, you have the opportunity to play one of your cards on this up card (which will then become the new up card) and thus reduce the size of your hand. The card you play, however, must be playable on the up card, according to the following rules:

- Most cards have a color -- red, green, blue, or yellow -- and you may play a card if it has the same color as the up card.
- The colored cards each have a rank -- either a number from 0-9, or else a special "skip," "reverse," or "draw 2" rank. You may play a card if it has the same rank as the up card, even if it is of a different color.
- There are two kinds of "wild" cards: ordinary wilds, and "draw 4" wilds. Either kind can be played on any up card. When you do so, you "call" a new color, specifying what color the next player must play.
- Draw 4 wild cards can only be played if the player has no cards of the same color as the up card in their hand.

Some special cards have an effect after being played, namely:

- If a "skip" card is played, the next player in sequence is skipped.
- If a "draw two" card is played, the next player in sequence must draw two cards from the deck and is then skipped.
- If a "wild draw four" card is played, the next player in sequence must draw four cards from the deck and is then skipped.
- If a "reverse" card is played, the sequence of players is reversed to counterclockwise (or back to clockwise, after an even number of reverses.)

The object of the game is to be the first to 500 points. You earn points by being the first in a round to run out of cards. When this happens, the player who has no more cards is awarded points based on the cards remaining in the opponents' hands. These points are calculated as follows:

- For every numbered card held by an opponent, the winner of the round gets points equal to that number. (5 points for a 5, no points for a 0, etc.)
- For every "special" colored card (draw two, reverse, and skip), the winner of the round gets 20 points.
- For every "wild" card (either normal or draw four), the winner of the round gets 50 points.

Normally, players continue playing hand after hand until one player reaches 500 points and is declared the overall winner of the game.

For this project, we will use the official rules with the following exceptions:

- We will not stop at 500 points. Since the computer is capable of playing hundreds of games per second, we will instead decide in advance how many games to play (perhaps 10,000) and decide the winner by who gets the most points totaled across all of them.
- If on their turn the player has a card in their hand that *can* be played, then they *must* play one. (It will be a rule violation to not play a card if a play is possible.)
- When a player goes out and the round ends, any draw 2 or wild draw 4 cards left in someone's hand will *not* result in drawing more cards to compute the number of points. Instead, draw 2 cards are worth 20 points and wild draw 4 cards are worth 50.

Before You Begin

- Make a copy of this Google Doc. You may put it in your shared folder if you like, but it is not necessary. (You will download a PDF of it and submit that for this assignment.)

  Go to this URL first to create your own version of the code in GitHub Classroom. Next, use GitHub Desktop to

get your own copy of the code (follow the instructions from these slides but look for your uno-YourUserID instead.

100 points

1: Scoreboard

The `Scoreboard` class

> Files to modify or use for this step:
>
> - Scoreboard.java - this is a Homework9 class you need to modify
> - `ScoreboardTest.java`

The `Scoreboard` keeps track of the names and scores of players in the game. When a scoreboard object is created, it's given an array of the names of the players; it should save that information and create space to store the score of each player. After each game is played, its `addToScore` method is called to award points to the player who won. Once the entire collection of games has been run, its `getWinner` method is called to determine who accrued the most points.

Fill in the following in `Scoreboard.java`:

1. any instance variables that are needed;
2. the constructor that's already started for you;
3. the getPlayerList and getNumPlayers methods;
4. the addToScore and getScore methods (following the comments);
5. toString, which should include each player's name with their score;
6. the getWinner method — but write pseudocode first (see below).

You should have written most of these methods in Homework 9 already.

When designing the `getWinner` method, work on pseudocode first and save your pseudocode in your Project Notes. If two or more players have the same score, it

doesn't matter which of those players your code chooses as the winner.

5 points

2: Card

The `Card` class

> **Files to modify or use for this step:**
>
> - `Card.java` - this is a class you need to modify
> - `Color.java`
> - `Rank.java`
> - CardTest.java

`Color` and `Rank` are enums that represent a card color — red, green, blue, or yellow — and a "rank," which is what kind of card it is (number, skip, reverse, draw two, wild, or wild draw four). The `Card` class describes a single card, having a rank and possibly a color and number.

The provided `Card.java` includes a skeleton class definition without instance variables. There are methods but they are just stubs without content. In this file you are to fill in:

1. any instance variables that are needed;
2. the three constructors already started; (be sure to initialize all three instance variables in each of the constructors)
3. the accessors `getColor`, `getRank`, and `getNumber` (you will *not* create mutators/setters; Card objects will be immutable, like Strings);
4. toString (should return a string such as "blue 8," "red draw 2," or "wild";
5. forfeitCost and isWildcard — as described by the comments;
6. canPlayOn — but write pseudocode first (see below).

Hint: be sure to read the comments in the Card class, as they provide detailed instructions about what the accessors should return for special cards.

Before you write the `canPlayOn` method, write down pseudocode in Project Notes (labeled `canPlayOn` pseudocode) for how to determine whether one card (the top) can be played on another (the bottom). (If the bottom card is a wildcard, there

will also be a "called" color that is the color the player decided it should act as.) Make certain you thoroughly understand the rules of the game before you do this, and express your pseudocode in terms of the instance variables you put in the Card class. Save your pseudocode in your Project Notes.

Run the unit tests in `CardTest.java` and make sure all of them pass before moving on to the next step.

10 points

Milestone 1: Code Review

Commit your java files using GitHub Desktop. I encourage you to commit your code regularly, not just once per milestone.

All the code up to this point should be completed by April 12th. You also need to schedule a code review meeting for the week of April 9th through April 12th (with Helen, by using this link). The purpose of the code review is to make sure you are progressing smoothly with the project and understand how to proceed, so that future milestones will go smoothly.

This part of the project cannot be late. Your grade will be based on whether you meet with Helen during the week and how many tests you pass when you meet with Helen that week to talk about your code.

5 points

3: EagerPlayer

Implement EagerPlayer

> Files to use for this step:
>
> - EagerPlayer.java — create this class
> - `Player.java` — the `Player` class; you will subclass this to create player strategies, deciding which card to play from their hand and which color to call when a wildcard is played
> - PlayerTest.java — abstract test class for players

- EagerPlayerTest.java — unit test for EagerPlayer

None of the above files should be modified. You should write your own EagerPlayer class.

Write a simple player strategy that extends the `Player` abstract class.

- `EagerPlayer`'s callColor method always chooses green, even if you have no green cards.
- `EagerPlayer`'s play method always chooses the first (lowest index) card that can be played. While the official rules state that a wild draw 4 card can only be played if there are no cards in the hand whose color matches the up card / called color, you can implement a slightly stricter (but easier to code) limitation that you only play the wild draw 4 if no other card in the hand can be played. To earn full credit, your `play` method should make use of methods that you already wrote in the Card class; answer the question in your *Project Notes Google Doc* before coding.
- `EagerPlayer`'s main method should be modified for debugging, by duplicating a test case that you failed (at one point).

You'll find it easier to implement callColor method before the play method.

10 points

4: Hand

Finish coding Hand

Files to modify or use for this step:

- `Hand.java` — one player's hand (an array of `Card`s) and the player object who will decide which card to play - this is a class you need to modify- **this latest version passes the checkstyle**
- HandTest.java — test class for Hand
- `Game.java` — the class that ties everything together; manages the turn-by-turn mechanics of the game
- `Deck.java` — the deck of Uno cards (as well as the discard pile)
- `Direction.java` — an enum for which direction play is currently going (needed due to reverse cards)

- `GameState.java` — information on what cards have been played and so on; player objects can optionally use this to perform more intelligent strategies (e.g. card counting)
- `Player.java` — the `Player` class; classes extending it have methods deciding which card to play from their hand and which color to call when a card is played
- `DeckExhaustedException.java` — needed for `Game` and `Deck` to compile

Now we complete the implementation of Uno so that the game can be played. Most of this code is already written for you; you will just need to fill in a few methods. For now, look over `EagerPlayer.java` and `Hand.java` and make sure you get the idea of what each one does.

The code you will write in this step will be focused on deciding which card should be played. Fill in the following methods in `Hand.java`:

1. `countCards` — determines the total point value of cards in this hand (used when some other player wins the game); your code should make use of `Card`'s `forfeitCost` method that you wrote earlier.
2. `selectColor` — asks the `player` which color should be called after a wildcard is played
3. `selectCard` — calls the `Player`'s `play` method to determine which card should be played. **This selectCard method should NOT implement a strategy** - instead, you should call the `Player`'s `play` method. To get the arguments of this method (e.g. the up card) you will need to call some accessors of the `game` object. Don't forget to call the removeCard method (from within the Hand class) to remove the card to be played from the Hand. (If there is no card possible to play — that is, `Player`'s `play` method returns `Player.NO_PLAY_POSSIBLE` — this method should return `null`.)

Hint: be sure to review your Lab 12 Notes and your Lab 12 code in replit to help you write the `selectCard` and `selectColor` methods. If you completed Lab 12 correctly, you will have `selectColor` almost completed, although the `selectCard` method will require some work.

Make sure to test your Hand class using the provided HandTest.java!

7 points

5: SingleUnoGame

Play a single Uno game

> Files to modify or use for this step:
>
> - `SingleUnoGame.java` — a class that sets up and plays a single Uno game - this is a class you need to modify

Now that you have coded a player strategy and the Hand class, you can play a game of Uno! Fill in the following code in `SingleUnoGame.java`:

1. any needed instance variables;
2. the constructor (already started for you);
3. the `getScoreboard` and `getPlayers` accessors;
4. the `playGame` method (which should create a `Game` object and call its `play` method).

After that, write code in the main method to create a `SingleUnoGame` object and call its `playGame` method.

- Set the public variable PRINT_VERBOSE (from the Game class) to true. This will allow you to watch each move played.
- Create two arrays (a String array of the players' names, and a Player array of EagerPlayer objects, to play a three-player game.
- Then create any other objects needed to set up the `SingleUnoGame` object.

Compile your code and run a game. Double-check that the game ran correctly, and fix any errors you discover!

All the code up to this point is due by Sunday, April 14.

5 points

Milestone 2

Milestone 2 is due by Sunday, April 14. Commit your code (whatever you have) using GitHub Desktop by April 14.

This part of the project cannot be late. Your grade will be based on how many tests you pass on April 14th.

3 points

6: SomewhatLessEagerPlayer

Code another player

> Files to use for this step:
>
> - SomewhatLessEagerPlayer.java — create this class
> - PlayerTest.java — abstract test class for players
> - SomewhatLessEagerPlayerTest.java — test class for `SomewhatEagerPlayer`
>
> You will need to write your own SomewhatLessEagerPlayer class.

Now you will create one more player strategy class, `SomewhatLessEagerPlayer`. This class needs to be completely written from scratch. It should extend `EagerPlayer` and override its methods as follows:

1. `callColor` — when a color has to be called, choose the color with the most cumulative point value in the hand ;
2. `play` — call the superclass (`EagerPlayer`) play method to select a card.

Don't forget the callColor method should never return Color.NONE. It should return the color out of the four other colors (RED, GREEN, BLUE, or YELLOW) that has the most cumulative points. (If the player only has Wild cards, call the superclass method instead.)

Hint: you can use the enum `ordinal()` and `values()` methods with the Color enum to simplify your code. Without these two methods, your code can still work, but it will be 4-5 times longer.

Be sure to consider how to properly leverage inheritance for these two classes. If you write working code that demonstrates you do not understand inheritance correctly, you will lose points on this step, even if your code works flawlessly.

10 points

7: Your Own Player

Make up your own strategy!

> Files to modify or use for this step:
>
> - Your Player Strategy — a class you started in Lab 12
> - `MultipleUnoGame.java` — a class that plays multiple Uno games and calcuates the best strategy - this is a class you need to modify

Now that you have the hang of making player classes, build your own! What strategy do you follow when playing Uno? What do you think would be the best? Each student's strategy will be tested against each other tournament-style at the end of the class, and the winner will be awarded full credit - including extra credit (105%) on the project!

Think about how you actually play the game of Uno: how do you decide what card to play? What color to call? Whether to switch the color or stick with it? When to part with a wild card? Try to write code that imitates your thought process. (If you're stuck for ideas, see below.)

5

MultipleUnoGame.java

I have provided you with `MultipleUnoGame.java`, which runs your `SingleUnoGame`'s `playGame` method many times. Customize the main method to plug in different players to give yourself a better idea of how your strategy performs:

- Change lines 98-103 of `MultipleUnoGame.java` to determine how the existing strategies perform.
- Run the program a few times to see how it works.
- Add code to the end of the main method to calculate the total number of points earned, and print out the percentage of points earned by each player. This is a good place to use printf instead of println (see pages 24-25 of your blue booklet), but you are not required to.

Go to your **Project Notes Google Doc** and fill in your findings. Answer questions 1-4 to help develop your strategy.

10

Your Strategy

To get full credit on your strategy and to compete in the tournament, you must attempt to implement an intelligent strategy for playing the game. You must strive to play not only a legal card, but a good card. I expect any intelligent strategy to beat EagerPlayer and SomewhatLessEagerPlayer, although you can still earn partial credit for an intelligent strategy that does not beat them (or only beats one of them):

- If your strategy plays an illegal card, you'll earn at most 3/10 on your strategy.
- If you develop a reasonable and valid strategy and explain your strategy well, but your strategy does not beat either EagerPlayer or SomewhatLessEagerPlayer, you'll earn at most 4/10 points.
- If you develop a reasonable and valid strategy and explain your strategy well, but your strategy consistently beats only one strategy (EagerPlayer or

SomewhatLessEagerPlayer), you'll earn at most 6/10 points.

- If you develop a reasonable and valid strategy, explain your strategy well, and consistently beat both strategies (EagerPlayer and SomewhatLessEagerPlayer), your strategy will earn at least 8/10 points (and possibly more points for performing well in te competition).

- If you beat both players and place in the Final Four of the Uno tournament, your program will automatically judged to be of high quality and you'll receive 10/10 points.

If you do not place in the Final Four, I will look at your well-commented, compellingly documented code to judge whether you attempted a non-trivial, intelligent approach. Your comments should narrate your algorithm completely and transparently. They should shed light on what your code is doing, and why, and explain the theory behind your approach. There is also a place in Project Notes where your strategy should be explained. I will award full points partially at my discretion based on how thorough and creative your program looks. Here are some strategy ideas that should earn full points:

- Maybe since your hand's points go to an opponent if you lose the round, you should try to minimize the number of points you hold, getting rid of wild cards as soon as you can, then the special cards, and then 9's before 8's, 8's before 7's, etc.

- On the other hand, it seems dumb to get rid of a wild card when you don't have to, since you can always hold it and play it later. So maybe you want to only play wild cards when you absolutely have to.

- Or maybe there's some middle "sweet spot" between ditching wilds too early and holding on to them too long and getting stuck with them.

- When you call a color, surely you want to call the color you have the most cards of.

- On the other hand, if you have green 0, 2, and 3, and red 8 and Skip, you have a heck of a lot more points represented in your red cards, so maybe it would be better to call red to get rid of those sooner.

- Suppose the up card is a red 5. You have both a red 3 and a blue 5. Should you change the color to blue or keep it red? This is similar to the decision about what to call when you play a wild, but with a twist: the up card may be red because one of your opponents wanted it to be red, and so it may be in your best interests to change it.

- Interestingly, there are fewer "0" cards than any other 1-9 number. (The standard deck has two "2s" and two "5s" and two "6s" of every color, but only one "0" of each color.) So playing a 0 card means it's less likely that an opponent will be able to change the color before you get a chance to play again. So maybe if you have multiple cards in a color, and one of them is a 0, you want to play that early on, even though it's worth less points, since you will be more likely to be able to get rid of another card in that color.

- Maybe all these decisions are affected by how big your overall hand is. If you only have a few cards left, you want to do everything possible to win the round

and claim the prize, even if this means taking short-term risks. If you have a ton of cards, on the other hand, you might want to forget about winning the round and simply ditch as many points as possible, figuring to lose as small as possible.

- Of course, your fellow students are reading these ideas too, so...maybe you should anticipate their following them and counteract. For instance, if everyone is going to try and ditch high point values first, you could try to keep high point values early in the game, so that it's less likely someone can switch the color on you later on by playing a card of the same rank. Or, if everyone is going to try to switch the color to favor their hand, you might want to deliberately call a different color when you have the choice, depending on circumstances.

Make sure to run unit tests on any Player classes you create. This will help ensure that your player doesn't inadvertantly cheat or otherwise break the rules due to a bug. You can do this by making a new test similar to the ones given to you in Step 6. The abstract PlayerTest class you're given does a pretty comprehensive set of checks; to run them on yours, you just need to extend it and pass your player object to the superclass constructor. This has already been completed for you in `MyPlayerTest.java` but you will need to make a copy of it if you decide to create multiple players to determine which is best.

You may find yourself creating multiple player files. If so, you'll need to create test files for each one. For example, if your player class is called BestPlayerEver, then you would make a BestPlayerEverTest in jGRASP in the usual way and put the following code in the class:

```
public class BestPlayerEverTest extends PlayerTest {

        public BestPlayerEverTest() {
            super(new BestPlayerEver());
        }
    }
```

This will automatically run the predefined tests in PlayerTest.

Any player classes that fail the unit test will automatically be disqualified from competition in the class tournament. Hence such a player class will not be able to give you extra credit.

Manual Grading

Some parts of the project will be manually graded, including a visual check that your comments are descriptive and that you have made good use of existing methods and inheritance throughout.

Tourney Play

The tournament will run as follows:

- First, take every possible combination of three players (including EagerPlayer and SomewhatLessEagerPlayer); each triad plays 10,000 Uno games. All 10,000 games are played on the same scoreboard, and at the end of the 10,000 games we decide on a winner (who got the most points) and loser (who got the least points).

- Keep the top four players; these are the final four - those four whose total number of wins minus number of losses is highest.
- Play 10,000,000 Uno games among the final four. The ranking of the final four is determined by how many points each has accumulated.

Intermediate and final results from the player tournament will be posted in Slack.

30 points

8: Reflection

Your pseudocode (for canPlayOn in Step 2) and your answers about the Game class are worth 5 points.
Your answers to questions 1-8 are worth 10 points.
If you have not done so yet, create a copy of this Google doc.

Project Notes (due 4/28): Canvas Link

Submitting the assignment

To submit this assignment, commit your java files using GitHub Desktop. The Project Notes will have its own Canvas link. I encourage you to commit your code regularly, not just once per milestone.

- Milestone 1 (due 4/7 - commit what you have to GitHub by 4/7, but you should continue to commit your fixes until 4/12):
  - upload two files: Card.java and Scoreboard.java
- Milestone 2 (due 4/14 - no extensions):
  - commit five files: Card.java, Scoreboard.java, EagerPlayer.java, Hand.java, SingleUnoGame.java
- Project Code (due 4/24):
  - commit eight files: Card.java, Scoreboard.java, EagerPlayer.java, Hand.java, SingleUnoGame.java, SomewhatLessEagerPlayer.java, MyPlayer.java, MultipleUnoGame.java
- Project Notes (due 4/28): Canvas Link