SDU

Assignment 2 in Object-oriented Programming

An Object-oriented UNO Game in Java.

Table of contents

1 Assignment 2 – UNO Game

Deliverables:

1. A concise report that documents your implementation using the provided template.
2. Java source code that implements the UNO game.

In this assignment, you need to demonstrate object-oriented principles by implementing a UNO game given the rules of UNO. The assignment includes OOP principles of abstraction, encapsulation, inheritance, and polymorphism. It is suggested that you read the UNO rules first ('*Appendix A: UNO Rules*'). Then, solve each subtask to eventually complete the whole assignment. **It is not required to solve all subtasks, but highly advised as it provides examples that you may use directly for the exam.** The subtasks are intended to be solved sequentially and vary in difficulty. The public interface of every class is given. Changes in the public interface should be carefully considered and documented. It is allowed to make any number of private methods to divide code within a class into smaller methods. Start by reading the UNO rules in '*Appendix A: UNO Rules*', since these have been slightly altered for simplicity reasons.

1.1 Subtask 1: Card and Numbered Card Classes

In this subtask, you need to implement classes that represent a card and a numbered card. Every card in UNO has a color, but not necessarily a number. Therefore, it is recommended to create an abstract class `Card` with a color attribute. The color may be represented by using an `enum` named `Color` with the following values: `WILD`, `RED`, `GREEN`, `BLUE`, `YELLOW`. Read sections 1-4 in this link to get familiar with how to work with enums. The UML class diagram is shown in Figure 1.

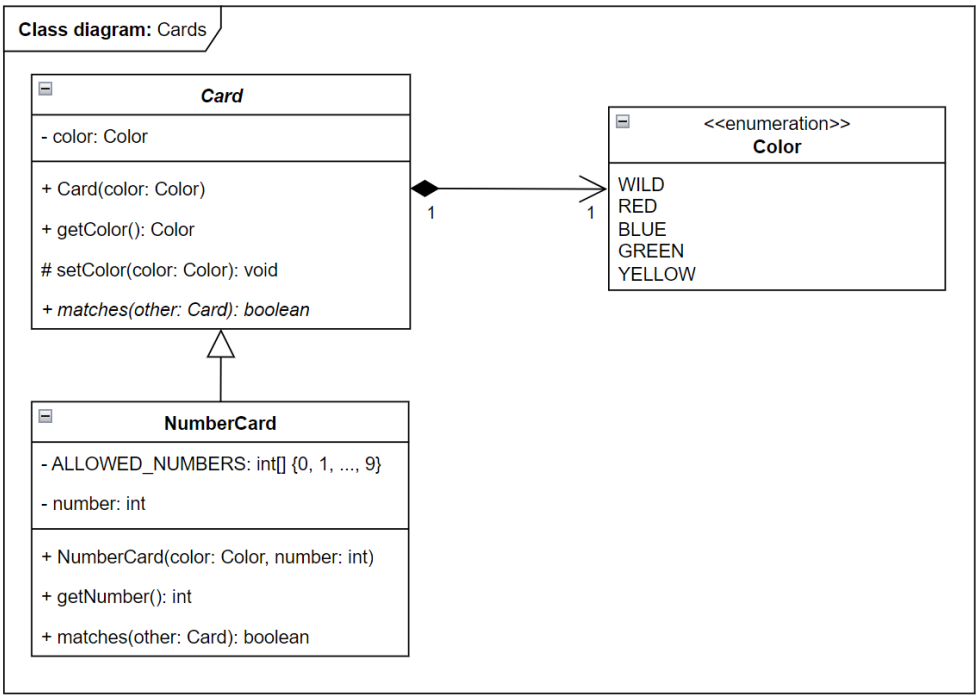


Figure 1. Card and NumberCard classes.

A card can only exist if it has a color. A color has no meaningful existence outside the `Card` class. Therefore, create an appropriate constructor. It must also be possible to retrieve the color through a get method. The set method must be accessible within subclasses. Use the appropriate access modifier for this encapsulation.

Furthermore, declare a public abstract method `boolean matches(Card other)` that will be used to check if the other card is a legal play.

The `NumberCard` class is a subclass of the abstract class `Card`. In addition to color, the numbered card has a number integer attribute. It also has a public constant of type int[] that contains the allowed numbers 0 to 9.

A numbered card can only be instantiated with a color and an allowed number. Create a constructor that matches this behavior.

The numbered card class has a get method to retrieve the number. Override the inherited `toString()` method to output a string similar to this format: YELLOW_8, BLUE_6, RED_0, etc.

Finally, override the inherited `matches(Card other)` method. Remember the rules of UNO; the card matches if either the color or number matches. *Hint: First check the color. Then use the '`instanceof`' keyword and casting to check the number*.

1.2 Subtask 2: The DrawPile Class

In this subtask, you need to implement the `DrawPile` class. A drawpile contains an arraylist of cards and is responsible for initializing a fresh pile of cards. Therefore, create a default constructor that initializes the card arraylist with the numbered cards stated in the UNO rules. Remember to shuffle the list. *Hint: Use Collections.shuffle(ArrayList)*.

Add a second constructor that takes an arraylist of cards as a parameter. This constructor is later used to initialize a drawpile from the cards in a discardpile.

The class provides a public method `draw()` that returns the drawn Card. The card may be chosen from either the front or the back of the arraylist. The `draw` method must throw a runtime exception if the pile is empty.

Add a public method `isEmpty()` that returns a `boolean` indicating whether this pile is depleted.

Add a public method `getSize()` that returns the number of cards within the pile.

1.3 Subtask 3: The DiscardPile Class

In this subtask, you need to implement the `DiscardPile` class. The discardpile contains all the played UNO cards of the game. Therefore, the discardpile has an arraylist attribute of type `Card`.

To construct a discardpile, an initial top card must be provided. A discard pile without any cards cannot exist. Create a constructor that takes a `Card` as a parameter. Remember to initialize the arraylist of cards, and then add the card to the arraylist.

Add a get method to retrieve the top card. The top card may be chosen from either the front or the back of the arraylist.

It must be possible to add cards to the discardpile. Create a public method `addCard(Card)` that returns a `boolean`. The method must call the card's

`match(Card)` method to check if it can be played against the top card. If it can, add the card to the discardpile and return true. Otherwise, return false.

Finally, the discardpile must provide a method that returns a new DrawPile object from the cards in the drawpile. This method is named `shuffleAndTurnAround()`. The `shuffleAndTurnAround()` method should (i) keep the top card within the discardpile, (ii) shuffle the remaining cards and (iii) return the shuffled cards as a new drawpile. Remember to clear the cards in the discardpile except the topcard.

1.4 Subtask 4: The Abstract Player Class

In this subtask, you need to create an abstract `Player`, and two subclasses; `ComputerPlayer` and `HumanPlayer`. A player encapsulates the attributes and logic of a UNO player. The abstract `Player` class is shown in Figure 2.

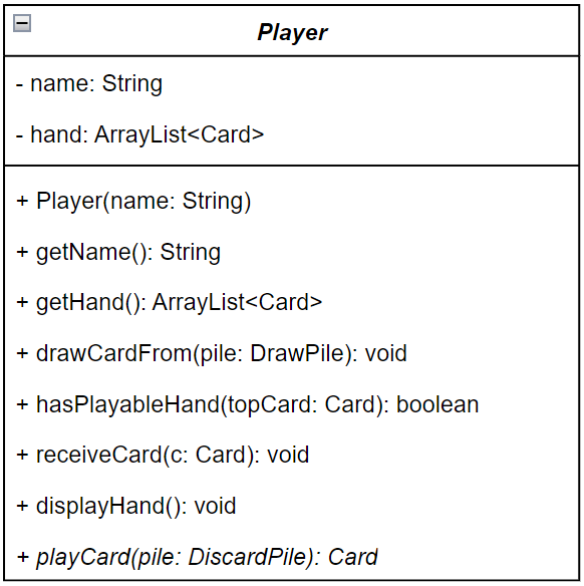


Figure 2. The abstract Player class.

A player has a name and a hand (arraylist) of cards. A player may only exist when given a name. This is reflected in the constructor. Given an instance of a player, it is possible to get the player's name and hand. This is reflected in the two get methods.

The `drawCardFrom(DrawPile)` method must draw a card from the drawpile and add it to the player's hand.

The `hasPlayableHand(Card)` must return true if the player has any card in the hand that matches the top card of the discard pile. Otherwise, the method returns false. *Hint: Invoke each hand card's `matches(topCard)` method to determine a match*.

The `receiveCard(Card)` method is intended for receiving cards when they get dealt before the game starts. Therefore, simply add the given card parameter to the player's hand.

The `displayHand()` method simply displays the content of the player's hand to the console. Here is an example output, but feel free to make your own format:

```
Bob's hand:
0:RED_5 | 1:RED_0 | 2:BLUE_9 | 3:GREEN_2 | 4:RED_1 | 5:YELLOW_7 | 6:YELLOW_4
```

The last method `playCard(DiscardPile)` is declared abstract because this behavior is left to either the `ComputerPlayer` or `HumanPlayer` subclass implementations.

1.4.1 The ComputerPlayer Subclass

Declare the `ComputerPlayer` class and extend the `Player` class. Make a constructor that matches the superclass with a name parameter.

Override the `playCard(DiscardPile)` method with a behavior like the following: (i) retrieve a playable card from the player's hand. *Hint: Use the card's matches(Card) method.* (ii) If the player has no playable card, throw a `RuntimeException` with an appropriate message. Otherwise, (iii) add the playable card to the discardpile and (iv) remove the card from the player's hand. Return the played card.

1.4.2 The HumanPlayer Subclass

Declare the `HumanPlayer` class and extend the `Player` class. Make a constructor that matches the superclass with a name parameter.

Override the `playCard(Discard)` method with a behavior like the following: (i) prompt the human player for a playable card from the hand. *Hint: Use the Scanner class to prompt for an index to select the card from the hand.* (ii) Prompt the user until a valid card is chosen. (iii) Add the selected card to the discardpile and (iv) remove the card from the player's hand. Return the played card.

1.5 Subtask 5: The UnoGame Class

Until this point, you've created the basic building blocks of a UNO game:

- `Card`
  - `NumberedCard`
- `DrawPile`
- `DiscardPile`
- `Player`
  - `ComputerPlayer`
  - `HumanPlayer`

But! The logic that binds it all together is still missing. Therefore, you need to implement the `UnoGame` class as shown in Figure 3.

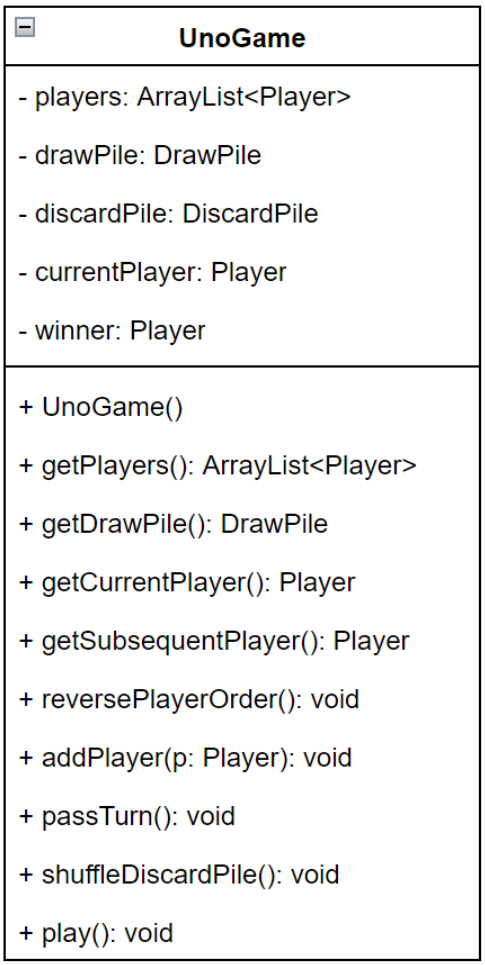


Figure 3. The UnoGame class.

The `UnoGame` class consists of an arraylist of players, a drawpile, a discardpile, the current player (an attribute that holds a reference to the current turn's player), and a winner.

Upon initializing the `UnoGame`, ensure to instantiate an empty arraylist of players, a new drawpile, and a new discardpile. Remember that the discardpile needs an initial top card. *Hint: Draw a card from the drawpile and use this as a top card.*

The methods `getDrawPile()`, `getPlayers()`, and `getCurrentPlayer()` simply return the corresponding class attributes.

The `getSubsequentPlayer()` method returns the subsequent player in the play order. *Hint: (i) Find the current player's index in the players arraylist and increment this value by 1. (ii) If the incremented index is larger than the number of players (minus 1), return 0. Otherwise, (iii) retrieve the player at the subsequent index.*

The method `reversePlayerOrder()` reverses the ordering of players. Example: The player order is player1, player2, player3. After invoking the method, the player order is player3, player2, player1. *Hint: Use the Collections.reverse(myList) method.*

The `addPlayer(Player)` method adds a new UNO player. The method must (i) ensure that a maximum of 10 players is allowed. Throw a `RuntimeException` if more players are added. (ii) The method must also check that the player does not already exist within the game. Throw a `RuntimeException` if the same user is added twice. (iii) Otherwise, add the player.

The `passTurn()` method simply overwrites the `currentPlayer` to the subsequent player. *Hint: Use the getSubsequentPlayer() method*.

The `shuffleDiscardPile()` method must create a new drawpile from the discardpile. But, (i) check that the drawpile is empty first. (ii) Overwrite the drawpile attribute by invoking the discardpile's `shuffleAndTurnAround()` method.

The `play()` method is the core of the UNO game. This method is built up of two major parts; (i) preparing the game and (ii) the game loop. Preparing the game consists of dealing cards (7 cards) to each player. Furthermore, a random player to start the game must be chosen.

The game loop essentially drives the game forward. It handles the core interactions of the game such as (i) printing the game status (the players, the number of cards in each player's hand, the top card, etc.), (ii) ensuring that a player draws a card if they have no playable hand, (iii) passes turn if no playable hand, (iv) makes a player select and play a playable card, and (v) pass the turn to the subsequent player. The pseudo-

code is shown in the listing below. Pseudo-code is a piece of non-executable code with the purpose of conveying the core principal ideas of that code.

```
1  // The UNO game loop.
2  play():
3      // Preparation section
4      if players < 2 or players > 10:
5          throw runtime exception
6
7      dealCards()
8      currentPlayer = randomPlayer()
9
10     // Core game loop section
11     do
12         printGameStatus()
13         drawIfNoPlayableHand(currentPlayer)
14         currentPlayer.displayHand()
15
16         if currentPlayer has no playable hand:
17             passTurn()
18             continue
19
20         playedCard = currentPlayer.playCard(discardPile)
21         print(playedCard)
22
23         passTurn()
24
25      while(gameNotFinished())
26
27     printWinner(winner)
28
```

Figure 4. The pseudo-code for the UNO game loop.

Lines 4-5 check that the UNO game contains the required number of players. Line 7 deals 7 cards to each player within the game. *Hint: Iterate the players and use a combination of the player's* `receive()` *method and the drawpile's* `draw()` *method.*

Line 8 selects and assigns a random player to start the game. *Hint: Use the built-in Random class.*

Lines 11-25 are the core game loop that is executed until a winner is found. Line 12 prints the status of the game. You decide what you want to print. It is suggested to at least print the top card, the current player, and the current player's hand. But print more information as needed. Line 13 checks if the current player has no playable hand and draws one card if this is the case. *Hint: Use the player's hasPlayableHand(topCard) method to check this. Remember to check if the drawpile is empty before drawing a card. If it is empty, invoke shuffleDiscardPile(). If not invoke the player's drawCardFrom(drawpile) method.*

Line 14 displays the hand of the player. Line 16-18 ensures to pass the turn if the current player has no playable hand.

Line 20 invokes the current player's `playCard(DiscardPile)` method and stores the returned card to display it in line 21.

Line 23 passes the turn to the subsequent player.

Line 25 checks the condition if the game is finished. The game is finished when the first player has no more cards in their hand. *Hint: Check every player's hand if it is empty. If a hand is empty, assign the winner attribute and return true. Otherwise, return false.* Line 27 prints the winner of the game.

1.6 Subtask 6: The Descriptable Interface

Each class in the UNO game should have a description. Declare the interface `Descriptable` and add the following method to the interface:

```
String getDescription()
```

Implement the interface in each of the classes (and subclasses): `Player`, `DiscardPile`, `DrawPile`, `Card`, and `UnoGame`. Reflect on why an interface is useful for this purpose. Would it be feasible to create an abstract class `Description` with an abstract `getDescription()` method instead and let every class in the UNO game extend this class? Why/why not?

1.7 Subtask 7: The Main Class

The `Main` class has a static main method that is responsible for instantiating (i) a new UnoGame, (ii) instantiating the players, (iii) adding the players to the game, and (iv) invoking the game's `play()` method.

See '*Appendix C - Example output*' for a sample run of the game. Note that your output need not look similar. You are free to make your own.

1.8 Subtask 8: Extending the Game with Action Cards

The game can be extended with action cards i.e., Skip, Reverse, Wild Card, Wild Draw 4 Card. In this subtask, you will demonstrate the strength of abstract classes.

Consider the UML class diagram shown in Figure 5.

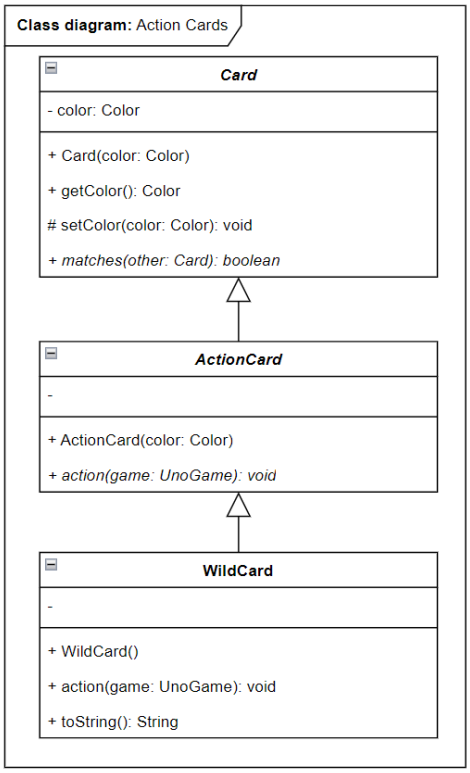


Figure 5. The extended Card hierarchy.

The UML class diagram shows that the abstract class `Card` is extended with another abstract class `ActionCard`. Furthermore, the `WildCard` extends the `ActionCard`.

An action card has a constructor with a color parameter. The color is the enum from earlier. The abstract method `action(UnoGame)` is intended to be overridden and act as a specific action card. With the `UnoGame` as a parameter, the action card is enabled to manipulate the game's state such as making the subsequent player draw +2 cards, skipping a player by invoking the game's `passTurn()` method, etc.

In this concrete example, you will implement the wild card. Remember the UNO rules; a wild card matches any color and requires the player to announce the color when played. The `WildCard` class will be responsible for handling this behavior.

Start by implementing the abstract `ActionCard` class given the class diagram. Then implement the `WildCard` class. The constructor of the wild card must always be `Color.WILD`. Call the super class constructor with this value.

Implement the public `setColor(Color)` method and call the super class' protected `setColor(Color)` method with the value.

Override the `matches(Card)` method by always returning true, since this card matches any other card regardless of color and number.

Before we can override the `action(UnoGame)` method, we need to add an abstract `announceCardColor()` method to the abstract `Player` class. This method returns a `Color`. This method will need to be implemented by the `ComputerPlayer` and `HumanPlayer` classes. Override the behavior of this method in the `ComputerPlayer` class to simply pick a random `Color` and return it. Override the behavior of this method in the `HumanPlayer` class to prompt the user for a color by using the `Scanner` class.

Now, override the wild card's `action(UnoGame)` method. The method must (i) invoke the `getCurrentPlayer()` method on the `UnoGame` class, and (ii) retrieve the color using the player's `announceCardColor()` method. Finally, (iii) print a message announcing the color, and (iv) invoke the card's `setColor(Color)` method.

Modify the uno game's `play()` method. The game loop needs to handle action cards. The loop can handle this abstractly by checking if the played card is an instance of an `ActionCard`. If it is, cast the card to an action card and invoke its `action(UnoGame)` method. Hint: This code needs to be inserted right after having invoked the player's playCard(DiscardPile) method in the game loop (line 22 in the pseudo-code). For the `UnoGame` parameter, use the '`this`' keyword.

Modify the `DrawPile` class' constructor to add 4 wild card instances to the drawpile.

Remember that according to the UNO rules, the first top card in the discard pile cannot be a wild card. Modify your code appropriately.

1.9 Subtask 9: To Infinity and Beyond

The UNO game can be extended further by adding all the remaining action cards: Draw+2, Reverse Card, Skip Card, and WildDraw+4. Here are some hints on how to do it. The regular action cards Draw+2, reverse card, and skip card can be implemented by extending the `ActionCard` class and overriding the `matches(Card)` and `action(UnoGame)` methods appropriately. All the regular action cards have a color, so they need to have a color set in the constructor. Remember to add a few instances of each action card class within the `DrawPile` class. It should not be necessary to modify the core game loop at this point.

The WildDraw+4 card can be implemented by extending the `WildCard` and overriding the `action(UnoGame)` method appropriately. Remember to add a few instances of the `WildDraw4` class within the `DrawPile` class. It should not be necessary to modify the core game loop at this point.

You can also implement custom action cards if you want.

1.10 Java Packages Requirements

Divide the classes into appropriate Java packages like the following example.

```
dk.sdu.<your initials>.uno
├─ cards
│  ├─ actioncards
│  │  ├─ ActionCard.java
│  │  ├─ Draw2.java
│  │  ├─ ReverseCard.java
│  │  ├─ SkipCard.java
│  │  ├─ WildCard.java
│  │  ├─ WildDraw4Card.java
│  ├─ Card.java
│  ├─ Color.java
│  ├─ NumberCard.java
├─ piles
│  ├─ DiscardPile.java
│  ├─ DrawPile.java
├─ players
│  ├─ ComputerPlayer.java
│  ├─ HumanPlayer.java
│  ├─ Player.java
Descriptable.java
Main.java
UnoGame.java
```

Table 1. Java packages structure.

2 Appendix A: UNO Rules

Table 2 shows the basic rules of the UNO game. Note that some rules from the original game have been altered slightly or removed for simplicity.

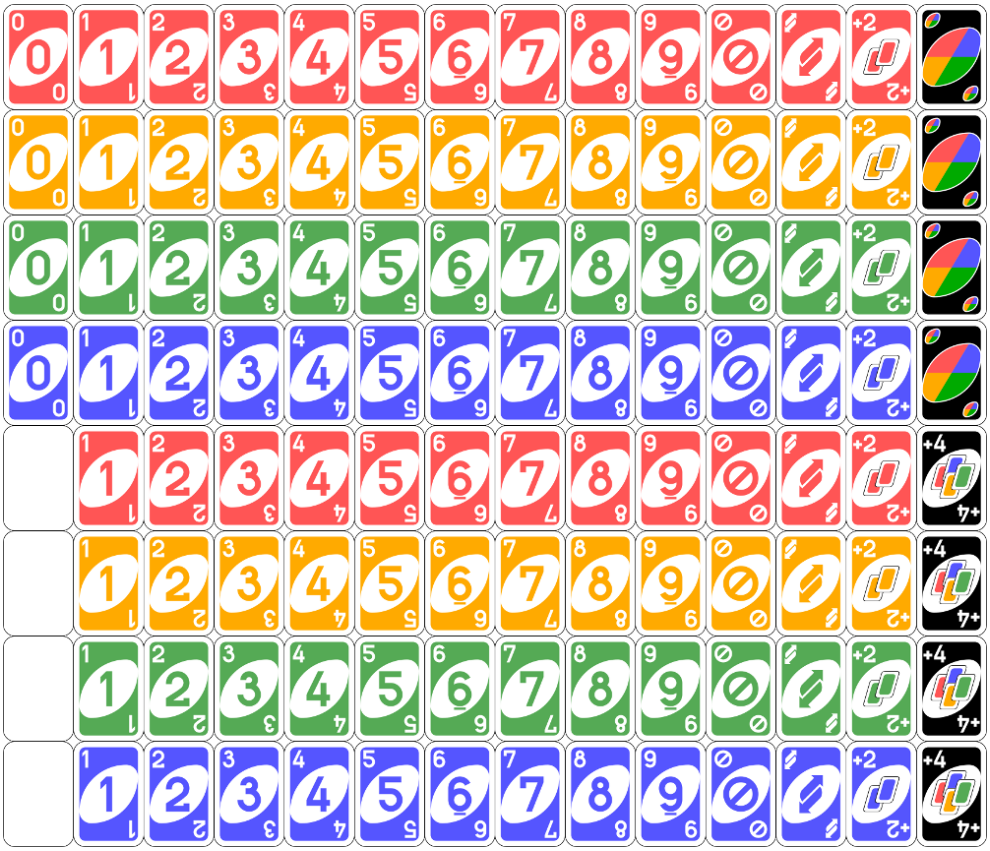
Cards	See '*Appendix B – UNO Standard Deck*'. **Numbered cards:** 19 Blue cards 19 Green cards 19 Red cards 19 Yellow cards The numbered cards have values 0 to 9. Only 1x 0 and 2x 1, 2, 3, …, 9 => 19 of each color. **Action cards:** 8 Skip cards – 2 each in Blue, Green, Red, and Yellow. 8 Reverse cards – 2 each in Blue, Green, Red, and Yellow. 8 Draw 2 cards – 2 each in Blue, Green, Red, and Yellow. 4 Wild cards 4 Wild Draw 4 cards
Objective	The first player to play all the cards in their hand wins the game.
Setup	The game has 2 to 10 players. The game has two piles of cards: • Drawpile • Discardpile Put all cards in the drawpile and shuffle them. Each player is dealt 7 cards from the drawpile. Turn over the topcard from the drawpile to begin a discardpile. If the top card is an action card turn over a new topcard.
Action cards	**Draw 2 card:** When you play this card the next person to play must draw 2 cards and forfeit his/her turn. This card may only

	be played on a matching color or on another Draw 2 card. **Reverse card:** This card reverses the direction of play. Play to the left now passes to the right, and vice versa. This card may only be played on a matching color or on another Reverse card. **Skip card:** The next person in line to play after this card is played loses his/her turn and is "skipped". This card may only be played on a matching color or on another Skip card. **Wild card:** When you play this card, you may change the color being played to any color (including the current color) to continue play. **Wild Draw 4 card:** This card allows you to call the next color being played and requires the next player to pick 4 cards from the drawpile and forfeit his/her turn.
Gameplay	A random player is selected to start the game. The order of play follows clockwise. The player must match a card in his/her hand with the top card on the discardpile either by number, color, or action card. For example, if the top card is a Green 7, you must play a Green card, any color 7, or Wild card, or Wild Draw 4 card. If you don't have anything that matches, you must draw a card from the drawpile. If you

	draw a card you can play, play it. Otherwise, the play moves to the next person. In regular UNO it is required to announce UNO when you have one card left in your hand. For simplicity, this rule is removed. Once a player plays their last card, the game is over.

Table 2. UNO Game rules.

3 Appendix B – UNO Standard Deck



4 Appendix C - Example output

```
+--------------------+-----------------------------+
|Drawpile: 54        | Discardpile: RED_9          |
+--------------------+-----------------------------+
|>Player: Alice      | Hand: 7                     |
|Player: Computer1   | Hand: 7                     |
|Player: Computer2   | Hand: 7                     |
+--------------------+-----------------------------+

Alice hand:
0:RED_9, 1:BLUE_4, 2:BLUE_9, 3:BLUE_1, 4:GREEN_6, 5:GREEN_5, 6:YELLOW_1,
Choose a playable card from the hand:
3
Choose a playable card from the hand:
2
Alice played: BLUE_9

+--------------------+-----------------------------+
|Drawpile: 54        | Discardpile: BLUE_9         |
+--------------------+-----------------------------+
|Player: Alice       | Hand: 6                     |
|>Player: Computer1  | Hand: 7                     |
|Player: Computer2   | Hand: 7                     |
+--------------------+-----------------------------+

Computer1 hand:
0:GREEN_6, 1:GREEN_2, 2:YELLOW_7, 3:BLUE_7, 4:BLUE_0, 5:BLUE_7, 6:BLUE_3,
Computer1 played: BLUE_7

+--------------------+-----------------------------+
|Drawpile: 54        | Discardpile: BLUE_7         |
+--------------------+-----------------------------+
|Player: Alice       | Hand: 6                     |
|Player: Computer1   | Hand: 6                     |
|>Player: Computer2  | Hand: 7                     |
+--------------------+-----------------------------+


Computer2 has no playable hand. Drawing.
Computer2 hand:
0:GREEN_9, 1:RED_4, 2:GREEN_5, 3:GREEN_3, 4:GREEN_0, 5:YELLOW_4, 6:YELLOW_8, 7:YELLOW_9,

Computer2 has no playable hand. Passing turn.

+--------------------+-----------------------------+
|Drawpile: 53        | Discardpile: BLUE_7         |
+--------------------+-----------------------------+
|>Player: Alice      | Hand: 6                     |
|Player: Computer1   | Hand: 6                     |
|Player: Computer2   | Hand: 8                     |
+--------------------+-----------------------------+

Alice hand:
0:RED_9, 1:BLUE_4, 2:BLUE_1, 3:GREEN_6, 4:GREEN_5, 5:YELLOW_1,
Choose a playable card from the hand:
```