Project 2: One Two Three Four I Declare Connect Fwar!

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Change Log

With a common lab assignment we do not typically have time to change the lab in response to student questions, concerns, and common misunderstandings. However, as this project lasts three weeks, it is relatively common for small updated, extra hints, and minor typos to be added. This page will list any such modifications. (and I promise we will have modifications as we go) **I recommend checking occasionally for updates on canvas and listening for updates announced in lecture**

- Version 1.0 2025-04-13 Week 2
  - Added clarity for to the ConnectFwar play functions.
  - Posted tester files for Board and Player.
  - Fixed tester for Card (small typo fix)
  - Added sections on testing and grading
- Version 0.9.1 2025-04-04 Week one
  - Added "getSuitNum" as a formally required function. This function is not as getRankNum, but is needed for our fancy toStrings and makes one of the Player objects a fair deal easier. Since it should be one line of code for most people, I figured let's just add it.
  - Posted tester files: CardTest, DeckTest
- Version 0.9 2025-03-27 Initial Version – Grading, testing, and submission details are still being decided upon and documented!

1.1 FAQS

None yet!

2 Essential Information

2.1 Pacing and Planning

This is a 3-week long assignment. As such you should expect it to be longer, and more complicated, than past assignments you have undertaken. Similar to last project – there will be several moments in this project where you need to make some key design decisions. As before – **plan time for understanding and designing code for this problem**.

Unlike the last project – this one focuses on *showcasing the power of object-oriented programming and inheritance* as such you will probably need meaningfully more total code than the last project. The upside of this, however, is that the code can, in principal, be well-structured to avoid any one function becoming too painful. If you can make good use of the provided design (and the flexibility for your own personal design) that this assignment provides, we hope you'll find this assignment longer, but not harder, than Project 1.

2.2 Deadline

The deadline for this assignment is Friday April 18th at 6:00PM. Late work is not generally accepted on projects – there will be a brief grace period on gradescope to help mitigate technical issues – but beyond this grace period late work will not be accepted without an exception. **Unlike labs there is no 1-day-late policy.**

2.3 Files

This project uses a fairly typical object-oriented design. This means that you will have many required files, but each file will not necessarily be super-big.

- `Card.java`
- `Deck.java`
- `Board.java`
- `Player.java`
- `ConnectFwar.java`
- `HumanPlayer.java`
- `BasicPlayer.java`
- `RandomPlayer.java`
- `RankRunner.java`
- `SuitStacker.java`
- `PlayerComparison.java`

Additional testing files may be provided over the course of the project, but ultimately you will be responsible for testing, and debugging, the code on your own.

2.4 Other important restrictions

Individual project – Unlike labs, where partner work is allowed, this project is an individual assignment. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in **a grade of 0 on this assignment**,

even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an **incident of academic dishonesty**.

To be very clear, you can ask other students questions only about this document itself ("What is Daniel asking for on page 3?"). Questions such as "how would you approach function X", "How are you representing data in part 2?", or even "I'm stuck on part B can you give me a pointer" are considered inappropriate collaboration even if no specific code is exchanged. Coming up with general approaches for data representation, and finding active ways to become unstuck are all parts of the programming process, and therefore part of the work of this assignment that we are asking you to do independently. Likewise, you are free to google search for any information you want *about the java programming language and included java classes* but it is not reasonable to look for information about the problem in this project (I.E. "how to program card games" would be an inappropriate google search)

Allowed java classes There are A LOT of java classes, which solve many interesting problems. You are not allowed to use any of the pre-built java classes except those we've explicitly discussed in class such as Scanner, Random, Math, and String. If you are unsure if a java class would be allowed here you should ask. Using unapproved java classes can lead to failing this assignment.

Specific examples of BANNED java classes:

- Any subclass of java.util.List (java's list classes)
- Any subclass of java.util.Map (java's dictionary classes)
- Any subclass of java.util.Set (java's set classes)
- java.util.Collections

Part of our goal here is practicing data storage using arrays. Every data storage task can be done by-hand without these advanced data structures.

3 Introduction

Over Spring break I was inspired to invent a new card game, a strange blend of several other games I had been thinking about lately. From Connect 4 I took the idea of placing things in columns to try to make a 4-in-a-row connection. From the card game War I took the idea of playing with just a shuffled deck of cards. And from a board game called "A gentle rain" I took the idea of having a singe payer game focused not on winner or loosing, but instead on getting the best score. Combining these pieces I came up with ConnectFwar – which will be described later.

As far as the game itself goes, ConnectFwar is fine. It's fun enough, but it's probably not the next "big thing" in solitaire card games. That said, every new game is an opportunity to learn – and in this case I was fascinated with a single question: **What's the best strategy for ConnectFwar?** If this were a "Classic" game I would turn to the internet – most games have been studied at length, but ConnectFwar is new, so we need a new approach.

My ultimate idea here is simple – First, I would program the basics of ConnectFwar. Then I could have the computer play the game tens of thousands of times with different strategies and see which one does best in practice. Finally, I could take the best strategy and adopt it's insights into my own gameplay to improve!

As a software project, this will have two core parts:

- Representing Card games (in general) and the ConnectFwar card game (in particular). This part of the project will focus on standard java objects and arrays.
- Representing different approaches to playing this game, and writing code to automatically compare strategies. This part of the project will focus on inheritance and polymorphism.

You can think of this as two "layers" to the code. First you'll need a "layer" simply to allow java programs talking about / thinking about the card game. Then you will write MORE core using this first "layer" of code, which will automatically compare strategies for our game.

3.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Implement several java objects of varying complexity.
- Make a few object representation decisions based on a general description of an object
- Practice using arrays for task-specific data structures (this will help prepare you for general-purpose data structures later!)
- See a specific fully-worked-out example where polymorphism and inheritance allow much more general and generic code.
- See how AI driven simulations can provide insight into real game playing.

4 Theory: The ConnectFwar card game

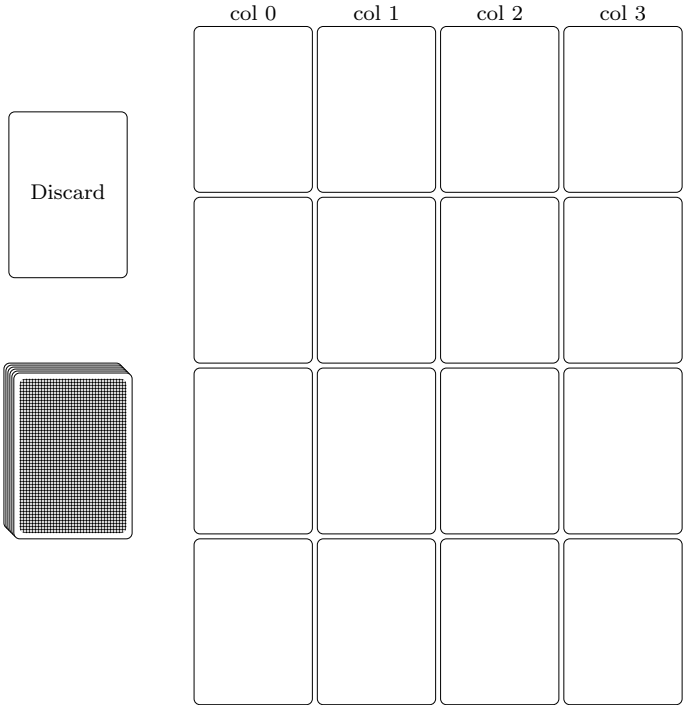
The purpose of this section is to teach the rules of the ConnectFwar Card game. *Very detailed* understanding of this game will be required to correctly implement it's rules into Java. I recommend reading this section carefully, and if you have cards, trying to play a game or two on your own to make sure you understand the rules. I also recommend **taking notes on this section**.

The ConnectFwar card game is a chaotic mixture of *Connect Four*, the card game *War*, with a few ideas stolen from the game *A gentle rain*. The end game probably doesn't look all that much like *any* of it's inspirations, but that's OK! We still thank the designers of all three games for inspiration.

ConnectFwar is a single-player game that requires nothing more than a standard 52 card deck of playing cards (Jokers removed) and a table to put cards on. A game of ConnectFwar involves drawing cards from the deck and then putting them down in a 4x4 grid with the goal of "connecting" 4 cards. The game ends when the last card is played, or when a connected four is made. While it is technically possible to lose a game of ConnectFwar, it is *exceptionally hard*. As such, don't think of this as a game you win or lose; instead, it's a game that you play for points – the more cards remaining in your deck when you connect four, the better.

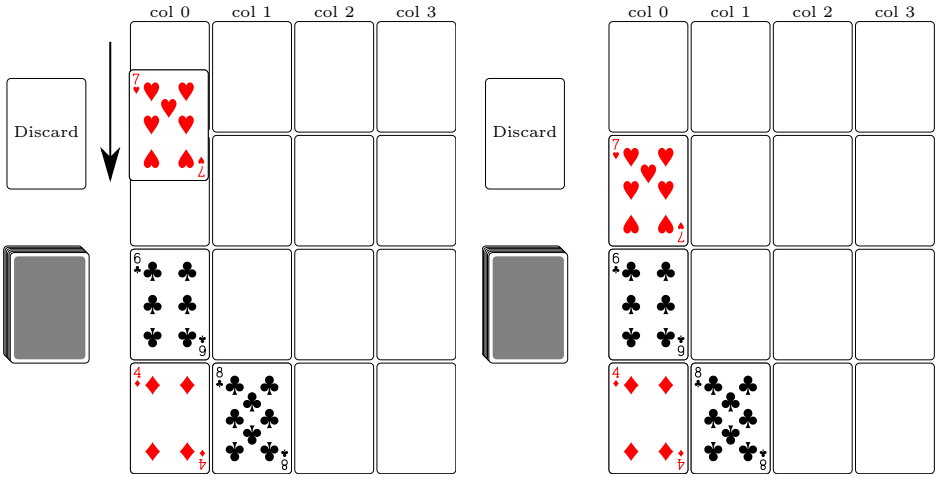
4.1 The core gameplay

The game starts by shuffling the deck of cards and setting aside space for a discard pile, as well as a 4x4 grid known as the "board". Each grid of the board can hold a card. This can be seen in the following image.

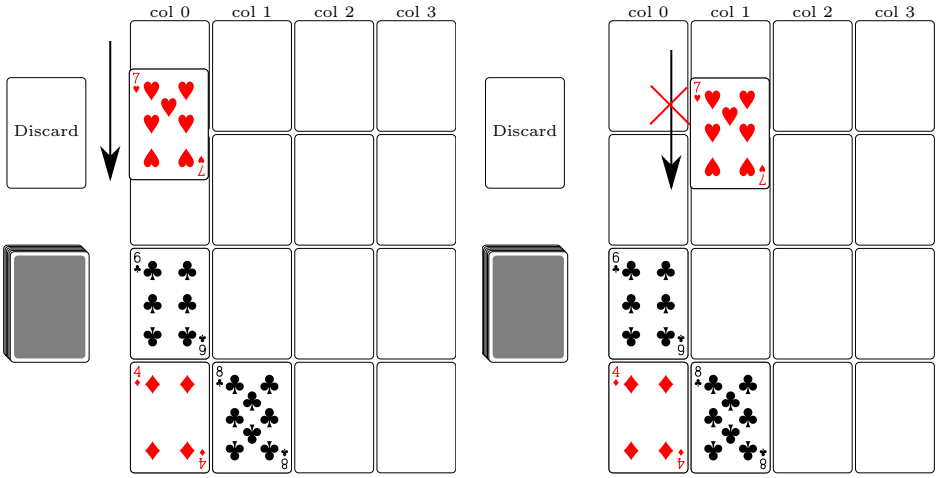


To play the game the player first draws a card. The player must then play the card to the board, or discard the card. To play a card to the board the player must pick one of the columns to

play. The card would then "sink" to the bottom of the column (like the tokens in connect 4!) So in the following example we're playing the seven of hearts to column 0. The card would "drop" to the third row from the bottom (because the first and second rows are already filled).



Once a card is played to the board it will stay there for the rest of the game. A card cannot be removed from the board, nor can it be "covered" by another card. The player can only play a card to a column if there is an empty space for it to go. For this reason, a player can play at most 4 cards to a given column, and a total of 16 cards to the board. Additionally, to place a card in a column, it must have the same, or higher value than the current top of the column. For example, in the previous image we could play the seven of hearts on column 0 because the current top of column 0 had value 6, but we could not have played the severn of hearts in column 1 since the top card had value 8.



For the purposes of ConnectFwar – Aces are the lowest value (treat them as having value 1), therefore they can only be placed on empty columns or other Aces. If there is no legal place to play a card, the player must discard it. The player is always allowed to discard a card, even if there is a legal place for it, the player is not forced to play any cards. After placing a card, a new card is drawn from the deck and the process repeats.

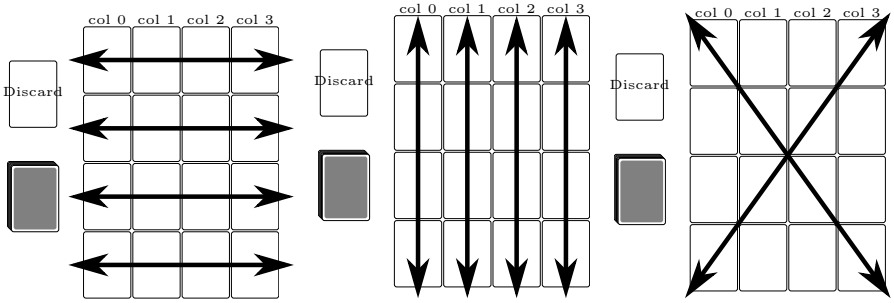
4.2 Ending the game

The game ends under 2 conditions:

1. The last card has been played/discarded without achieving a "connected four" and there are no more cards to draw. This is said to be a loss and is worth 0 points. (I should note – this is not common.)
2. When there is a "connected four".

Note: technically, for "human" play – we would normally list a third ending – the board is full (said to be a 0 point loss). We do not list this as we don't *technically* need this case. If the board is full there are no legal plays, so the player would be forced to discard until they lose anyways. That said, most human players adopt this rule as a simple time-saver.

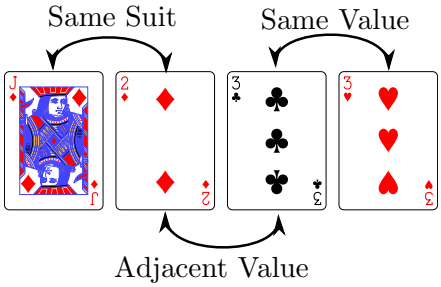
A connected four can occur on any row, any column, or either of the diagonals of the board:



For four cards to be considered "connected" each adjacent pair of cards needs to have one of:

- The same suit
- The same value
- Adjacent values (next biggest or next smallest)

Note – we do not require each pair in the run of 4 to have the same type of connection. For instance, the following is a valid series of connected cards:



The game ends when a connected four is created. Technically, the game ends when the connected four is first noticed – with human players it's not unheard of to "miss" a connected 4 like the previous example. However, for digital play we will simply check for a connected four after each card is played and end the game as soon as one is created.

As ConnectFwar is a solitaire game, and one that's hard to lose, the goal is not to win, but instead to score highly. The score at the end of the game is based on how many cards are left in the deck. A score of 48 is the best score, and means that the player got a connected four with the first four cards they drew. The lowest score is 0 and can happen either by winning on the last card, or simply running out cards without connecting four cards.

4.3 Summary

So a game of ConnectFwar would look like this:

1. Clear some space on a table for the board
2. Shuffle the deck of cards
3. Until the game is over
   (a) Draw a card. If you can't because the deck is empty, then the game is over and you've lost.
   (b) Place this card in one of four columns (following the rules for legal placement) or discard it.
   (c) Check for a connected four – if a row, column, or diagonal has a connected four, the game is over.
4. Count the remaining cards in the deck – that is your score.

Take a moment to review this outline You will ultimately need to program this logic. Make sure you understand the details, and maybe even take out a deck of cards and try to play it. This thinking will come in useful later – so take notes now while the design of the game is fresh in your mind!

4.4 ConnectFwar strategy?

While AIs for single player games are less common, exploring them is still a good easy way to learn what works and what doesn't for your personal play. As such, we will be exploring 4 key strategies for this game:

- Random play – pick a random column (regardless of validity)
- Basic – pick the first valid column (left-to-right)
- Suit Stack – Put each suit in it's own column (playing any time it's legal)
- Rank Stack – Try to put each card rank in it's own columns. Since there are more ranks (13) than columns (4) we will have to "share" stacks.
  - We will do this with the equation $(rank - 1) \mod 4$. This might sound complicated, but it's rather straightforward in practice:

A	2	3	4	5	6	7	8	9	10	Jack	Queen	King
0	1	2	3	0	1	2	3	0	1	2	3	0

These are ultimately pretty basic algorithms, but even these will tell us something about what works, and what doesn't work, in this game. You are, of course, invited to try to come up with better tactics than this.

5 Requirements

The design for this program is very typical of an Object-Oriented design. This is to say, it has many classes, most of which are small and serve one well defined purpose:

- `Card` - represents one playing card
- `Deck` - represents a deck of playing cards
- `Board` - represents a state of the board, also containing logic around what plays are legal, and checking if the game has finished.
- `Player` - An interface representing what information the game needs from a player (namely, which column they play a card to)
- `ConnectFwar` - A class with static functions for playing a game of ConnectFwar with a given Player object
- `HumanPlayer` - An implementation of player that uses `System.out` and `Scanner` to get moves from a human player.
- `BasicPlayer` - An implementation of Player that takes the first legal move left-to-right
- `RandomPlayer` - An implementation of Player that takes moves at random
- `RankRunner` - An implementation of Player that puts each rank in it's own column
- `SuitStacker` - An implementation of Player that puts each suit in it's own column
- `PlayerComparison` - A class with static functions for testing the AIs, reporting their average score.

While this may sound like many classes to write, most of them are relatively simple on their own (several are about 4 lines long, including the class definition). By splitting the behavior up like this you can also focus on single classes at a time. You could, for example, try to write one (and only one) class a day and be done in a week and a half.

5.1 Card

The `Card` class represents a single playing card. There are MANY ways to represent a playing card in any modern programming language. We are going to represent these with two integers: `rank` (the number on the card) and `suit`. For rank we will use 1 to represent "Ace", 2 to represent "Two", and so-forth until we hit 11 (Jack) 12 (Queen) and 13 (King). For suit we will use 1 to represent Spades, 2 to represent Hearts, 3 to represent Clubs, and 4 to represent Diamonds.

Your card object should be immutable, meaning that once constructed there should be no way to change it. Your Card class can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names and parameter types (although you can name the parameters differently).

- `public Card(int rank, int suit)`
  Constructor - the first `int` should indicate the rank of the card (1 = Ace, 2 = Two, ..., 11 = Jack, 12 = Queen, 13 = King). The second `int` indicates the suit 1 = Spades, 2 = Hearts, 3 = Clubs, 4 = Diamonds. This constructor should validate it's inputs – if an invalid suit or rank is given it should print an error message. ("Invalid Card") and then set private variables to be the Ace of Spades.
- `public int getRankNum()`
  This method should return the number representation of the cards rank.

- `public int getSuitNum()`
  This method should return the number representation of the cards suit (similar to the constructor parameter).
- `public String getRankName()`
  This method should return the string naming the cards rank. (I.E. "Ace", "Four", "Ten", "Queen" etc.) All rank names should be capitalized
- `public String getSuitName()`
  This method should return the string naming the cards suit ("Spades", "Hearts", "Clubs", or "Diamonds")
- `public String toString()`
  Your Card should override the default toString method to one that prints a human readable name for the card. Once written this will help you when using print to debug since it provides a human readable description of the card. Examples of the type of output we are looking for can be found in the test files.
- `public boolean equals(Object obj)`
  You card class should override the default equals method. A Card should only be equal to other instances of the Card class, and then only other cards that have the same rank and suit. **note** This is not a typo – the parameter should be Object, not Card here. We will discuss this function, why it is the way it is, and how to program it, in lecture.

5.1.1 Additional Card Functions

We have prepared two additional card functions `toFancyString` and `toVeryFancyString` These can be found in the Card.java file provided on canvas. These functions prepare strings that use Unicode letters that may not look correct on all computers, and may not behave reliably on all computers, but provide much more compact representations of the cards. You should not need to modify or even substantially understand these functions, but you are required to include them. `toVeryFancyString` should produce letters that occupy 2 spaces in the terminal, and look like tiny playing cards. `toFancyString` should produce short-names for each card, using unicode letters for the suit symbols.

If neither of them work, don't worry too much – in a pinch you can always edit toFancyString to use only "normal" letters.

5.2 Deck

The `Deck` class represents a deck of cards. It must use an array of type cards (length 52) to represent the cards and their current order. You will also need at least one additional variable to track which cards have been dealt, and which have not.

NOTE/WARNING I'm giving you extra design freedom for how to make the private internal variable management of this class work. You are required to store cards in an array, but beyond that we are not telling you how to track which cards are used and which are not. As such, many of the function descriptions are written *abstractly* representing the external view of the object, not it's internal view. For example, the *draw* method will say it "removes a card from the deck" – since you are storing your deck in an array you likely *will not* actually remove the card from the array, but instead would update variables so that this card cannot be drawn again. You should seek an efficient way to do this, and think carefully about the $O(1)$ efficiency requirements below. As a hint – one integer, used well, can solve this problem.

Your `Deck` can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names, and parameter types (although you can name the parameters differently)

- `public Deck()`
  Constructor - creates a new deck. Makes an array containing 52 different Cards. You must use one or more loops: you will receive no points if you just write 52 assignment statements. The order of Card's within the array does not matter. The last line of your constructor should call your shuffle method so that all decks are shuffled by default.
- `public void shuffle()`
  Shuffle the deck of Card's that is represented by the array you made in the constructor. The easiest way is the Durstenfeld-Fisher-Yates[1] algorithm, named after its inventors. The algorithm exchanges randomly chosen pairs of array elements, and works in $O(n)$ time for an array of size n. You must use the following pseudocode for this algorithm.
  Do the following steps for the integer values of i starting from the length of the array minus one, and ending with 1.
  1. Let j be a random integer between 0 and i, inclusive.
  2. Exchange the array elements at indexes i and j.

  (To generate random numbers, please use Java's Random object, and in particular the Random class method `public int nextInt(int bound)` which generates a number between 0 and *bound* (not including *bound*)).
  Shuffling a deck that has been dawn from should reset it as if it was a new deck, All 52 cards should be available again, and the cards should be in a new random order.
- `public Card draw()`
  Draw and return the next card. Whatever card is drawn should not be drawn again until the deck is shuffled again. This should decrease the number of cards remaining. Note you do not want to pick a card at random here – the `Card` array should be in a random order, so you should come up with a way to return the "next" card that has not been drawn. This method must work in $O(1)$ time. If the deck is empty it should return Null.
- `public int cardsRemaining()`
  Returns the number of cards remaining before the next reshuffle. This should return 52 after constructor or a call to shuffle, and decrease with calls to draw down to 0.
- `public boolean isEmpty()`
  Returns whether or not the deck is empty.

5.3 Board

The `Board` class represents the game board for ConnectFwar. This class is in charge of tracking the 4-by-4 grid of cards, knowing the rules for what cards can be played where, as well as the rules for determining if the game has ended with a connected four.

The `Board` class must internally store a 2-dimensional array of cards. An empty array of this sort can be created with the line `this.cards = new Card[4][4];`. Technically this makes an array

[1]For more information on this algorithm I recommend the Wikipedia page it is both informative, and currently free of actual Java source code (looking up source code for this algorithm would be cheating, so be careful what you search)

of arrays, and also fills the four cells of this primary array with 4 other arrays. You can basically treat it like a 2-dimensional array like so: `this.cards[1][2]`.

You do not need other private variables than this, but you are free to define them if you want. You must implement the following pubic functions, you are free to implement more if you want.

- `public Board()`
  Constructor. ths should create an empty 4-by-4 board.
- `public Card getTopCard(int column)`
  Get the top card of a given column. Columns are zero-indexed. If no card has been played to a given column, return null, otherwise return the card in the highest position in that column.
- `public Card getCard(int col, int row)`
  Return the card at a given position (indicated by column number and row number). Rows are zero-indexed, with row 0 being the bottom of the board. If there is no card there, return null;
- `public boolean canPlay(Card c, int column)`
  This should return a boolean to indicate if it would currently be legal to play a given card to a given column. If the column number chosen is not valid – return false.
- `public void play(Card c, int column)`
  This should update the board to show the result of playing a card to the given column. Note – the behavior of this function is *unspecified* if the play is not legal. That is to say, you can choose whatever behavior you want for this situation – it shouldn't matter because other functions shouldn't attempt to make an illegal play.
- `public boolean isWinState()`
  This should return a boolean to indicate if the game currently has a connected four.
  - Remember that this needs to check all 4 columns, all 4 rows, and both diagonals of the board.
  - While you are free to use helper functions to decompose any task, you are specifically encouraged to break this task down. While it's possible to do all this work in one big function, such a function would be *repetitive* and *hard to fix or change*.
  - If you're having trouble thinking of ways to decompose this problem one great place to *start* is with a function to confirm that two card are "connected" (same suit, same rank, or adjacent rank).
- `public String toString()`
  This should create a string representation of the board. This can look a few ways depending on whether you use the provided `toVeryFancyString` or `toFancyString` function.

5.4 Player

The player Interface should be quite simple: It only needs to specify a single function:

```
public int nextMove(Card card, Board board)
```

As `Player` is an interface, not a class, remember that you wouldn't be implementing this function here. That said, we will describe the parameters, return value, and general role of this function here – since this is shared by the 5 required implementations of this function.

This function will be called by the `ConnectFwar` class as part of the core logic of the gameplay. It is called after drawing a card from the deck to decide what column to play the card to. The Card parameter is the card that was drawn from the deck, the Board parameter represents the current game board, and the returned integer represents the player's choice of columns. If the function returns an invalid column (either outside of the 0-3 range, or somewhere the given card cannot legally be played) then the ConnectFwar code will assume this Player intended to discard the card. As such, this function is allowed, and in some cases expected, to return values outside the 0-3 range of valid columns.

5.5 ConnectFwar

This class has only one required function:

```
public static int play(Player player)
```

This function should play a full game of ConnectFwar using the Player object to decide what move to make each turn. This game needs to implement the whole game workflow, including creating a deck and a board for the game. Remember, the `Player.nextMove` function may return an invalid column, so you will need to check if this is a valid move before playing the card.

The return value for this function is the score earned when playing the game. (Remember, you don't really win or lose ConnectFwar – you play for points)

5.6 HumanPlayer

This class should implement the Player interface allowing a human to play the ConnectFwar game. This method of interaction will not be pretty as it's intended for testing and development, not being used by the general user.

This class should have a private `Scanner` variable. We will need to use this to get choices from the user in the `nextMove` function. We want this to be an instance variable, rather than local to

the `nextMove` function to avoid the complexities that come with making multiple `Scanner` objects for the single `System.in` object.

This class has two required functions:

`public int nextMove(Card card, Board board)`

This should print the card function's toString (Fancy or normal, your choice), then print the board's toString function, and then finally use scanner to get an integer from the user, which it can return directly without validation. Note – you should have a private scanner variable, NOT create a new Scanner each time this function is called (Scanner objects are not intended to be repeatedly created like that).

`public static void main(String[] args)`

This function should allow you to play a game of ConnectFwar using the HumanPlayer.

Note Ultimately, this class doesn't directly serve our end-goals, but it will substantially help you understand the game and test for issues with your game. As such we decided to make it a formal requirement. Once you get this done, take a moment and play a few rounds of ConnectFwar, you've earned it!

5.7 BasicPlayer

The BasicPlayer class implements the Player interface. This AI serves as a simple baseline, just making the first legal choice it sees.

`public int nextMove(Card card, Board board)`

This function should check each column in order, returning the first column that you can legally play the card to. If the card cannot be played than any integer can be returned (as this card must be discarded regardless).

5.8 RandomPlayer

The RandomPlayer class implements the Player interface. This AI serves as a simple baseline, just acting randomly.

`public int nextMove(Card card, Board board)`

This function should return a random column. Ths function is not expected to check if the random column is valid – if an invalid column is randomly chosen, the card will be discarded.

5.9 RankRunner

The RankRunner class implements the Player interface. This AI attempts to make columns of cards with the same number, or rows of cards that are increasing/decreasing in value.

`public int nextMove(Card card, Board board)`

This function should return 0 for all aces, 1 for all twos, 2 for all 3s, 3 for all 4s, 0 for all 5s, 1 for all 6s etc. (This can be done rather easily using the rank number, the modiolus operator (`%` and a little bit of math)) This function is not expected to check if the play is valid.

5.10 SuitStacker

The SuitStacker class implements the Player interface. This AI attempts to make columns of cards with the same suit.

```
public int nextMove(Card card, Board board)
```

This function should return the card's suit number minus 1 – in this way each suit will be matched to it's own column. This function is not expected to check if the play is valid or not.

5.11 PlayerComparison

This class contains a main method (and any helpful private functions) that will report the average sore earned by each of the 4 AI players we built. You can compute this average by having each AI play 10,000 games, adding up the scores, and dividing by 10,000. The output of my program is given below: Your code does not need to output this exactly (in fact, I would be surprised if it did, as estimated scores are based on the random shuffling) but it should contain all this information. You're free to format this information somewhat differently, so long as your format is clear.

```
random 12.8218
basic  30.1236
suits  23.9441
ranks  26.8736
```

6 Testing and incremental development

This project is quite typical of an object-oriented program design: There are A LOT of things to program, many of which are individually quite small. The idea here is to spread the complexity of the program over more code – that way no one function is individually super-complicated. It might seem weird on the surface, but it is FAR easier, and FAR faster to program ten 15-line functions than it is to program one 100-line function. This leads to longer development time, but often faster testing, less debugging, and easier programming in general.

We have the following test files:

- `CardTest.java` – relatively comprehensive tests for the Card class.
- `DeckTest.java` – relatively comprehensive tests for the Deck class.
- `BoardTest.java` – mostly comprehensive tests for the Board class.
  - There are A LOT of possible things to test around the isWin condition. This performs a few basic tests, but we will largely plan on code reading for this. As such make sure your code is readable.
- `PlayerTest.java` – basic tests for the player classes

Note that this does not cover all the behaviors we have required you implement in PDF and will be graded for. The following Manual checks should be expected, and probably should be done manually.

1. Code review – all required functions exist, are readable, are obviously correct (I.E. it's easy to see why they are correct) and behave as expected
2. Code review – all code style guidelines as discussed throughout the semester are followed.
3. Code review – a specific review of the Board isWinState function will be needed.
4. Code review – Player is an interface
5. Code review – HumanPlayer has a Main method
6. Code execution – The HumanPlayer main method runs a game suitable for basic testing/debugging/
7. Code review – Random player plays randomly with no regard for validity of moves.
8. Code review – Basic player doesn't crash when given cards that cannot be played.
9. Code review – the ConnectFwar play method correctly describes the process of playing a game.
10. Code review – the PlayerComparison class does as specified
11. Code execution – The PlayerComparison class main method runs and produces reasonable looking output.

7 Submission

For submission make sure you turn in at least:

- `Card.java`
- `Deck.java`
- `Board.java`
- `Player.java`
- `ConnectFwar.java`
- `HumanPlayer.java`
- `BasicPlayer.java`
- `RandomPlayer.java`
- `RankRunner.java`
- `SuitStacker.java`
- `PlayerComparison.java`

8 Grading

Grading is still approximate until we get the autograder written, but it will be approximately like this.

- (around 16 points) Autograder
- (10 points) Code style
- (6 points) Manual code execution tests
- (8 points) Card class
- (8 points) Deck class
- (20 points) Board class
- (18 points, 3 each) Player interface and it's 5 subtypes
- (10 points) ConnectFwar class
- (4 points) PlayerComparison Class