CPSC 231: Computer Science II Sections 4 & 5
Spring 2020
Mastery Project: Modern War(fare)
Due: May 4, 2020 at 11:59 pm

Reading
This assignment assumes you have completed the reading for weeks 1-10 on zyBooks.

Being Upfront...Up Front
We are well aware that this assignment and its solution has been disseminated widely on the internet on sites such as Chegg.com. Last semester we had to penalize some individuals for copying code from these solutions. In cases where these solutions were not given attribution, this involved formal reports to the academic integrity committee.

For this assignment, you are prohibited from searching for, looking at, or sharing any code available publicly or privately that is related to the solution for this assignment. We will be using MOSS (http://theory.stanford.edu/~aiken/moss/) to check all code submitted for this assignment against solutions available online as well as submitted during the previous 3 semesters. Any evidence of copied code from these sources will result in an academic integrity violation with a grade of "F" in the course. Because the instructions for this assignment explicitly prevent referring to such sources IN ANY CAPACITY, we will take action even if this code is given proper attribution. Thus, if you reference sources for this assignment, you are better off not submitting your work and receiving a 0 than you are submitting work that resembles previous solutions.

Note that this assignment has also been changed from previous semesters, so it will be immediately obvious if previous solutions are copied.

You MAY: Google how to do some specific thing in Java that isn't related to how-to-write-war-card-game stuff, look at the Java API, and consult with me for help.

The Assignment
Now that you know how to write OO Java code, let's put your skills to use in your first large programming assignment. Your task for the next week is to write a simulation of a modified version of the card game War. We call it...Modern War(fare).

How to Play
In this game of War, two players battle each other to see who can win the most cards. At the start of a single game a deck of 52 cards (legal values 2-10, J, Q, K, A) is split evenly and randomly between the two players such that each player holds a stack of 26 random cards.

In terms of card value, **SUITS DO NOT MATTER.** A 4 of Diamonds has the same value as a 4 of Spades – 4.

For each battle of the game, both players turn over three cards at the top of their stack. Each player picks the middle value card and compares the value with the other player**.** So if a player turned over a King, a 4, and a 6, the middle value card would be the 6, and the player would use the 6 when comparing cards with the other player.

If a player has fewer than 3 cards, he or she uses the maximum value card. For example, if a player only has two cards left and puts down a 5 and a Jack, he/she would use the Jack to compare against. If a player only has one card left, he/she just uses that card to compare against.

The player with the higher value takes all cards played in the battle and places them on the **bottom** of their stack.

If the two values played are equal (a "war"), each player lays down one card. The higher-valued card wins all of the cards on the table, which are then added to the **bottom** of the winning player's stack of cards. In the case of another tie, the war process is repeated until there is no tie. If a player runs out of cards during a war, the other player wins. (If both players run out of cards at the same time, it's just a giant shitshow and pretty much the world ends and really tbh we don't want that to happen. But yeah, if it happens, no one wins and the game ends and whoever had the most cards before the war wins the game.)

A player wins by collecting all the cards. Once a single player holds all 52 cards, the game ends.

The Goal

You are to write a program that takes the number of games, *n,* to play as a command line parameter (get passed in the String[] args parameter to main, not from standard input) and then simulates the *n* games. At the conclusion of all the games, your program will calculate and output the following statistics:

Average number of battles per game
Average number of wars per game
Average number of double wars per game
Max number of battles in a game
Min number of battles in a game
Max number of wars in a game
Min number of wars in a game

Designing a Solution

As you can probably tell, this assignment is fairly involved. In fact, the only way you can pull it off is if you are smart about the abstractions you use and how you break your program into separate classes.

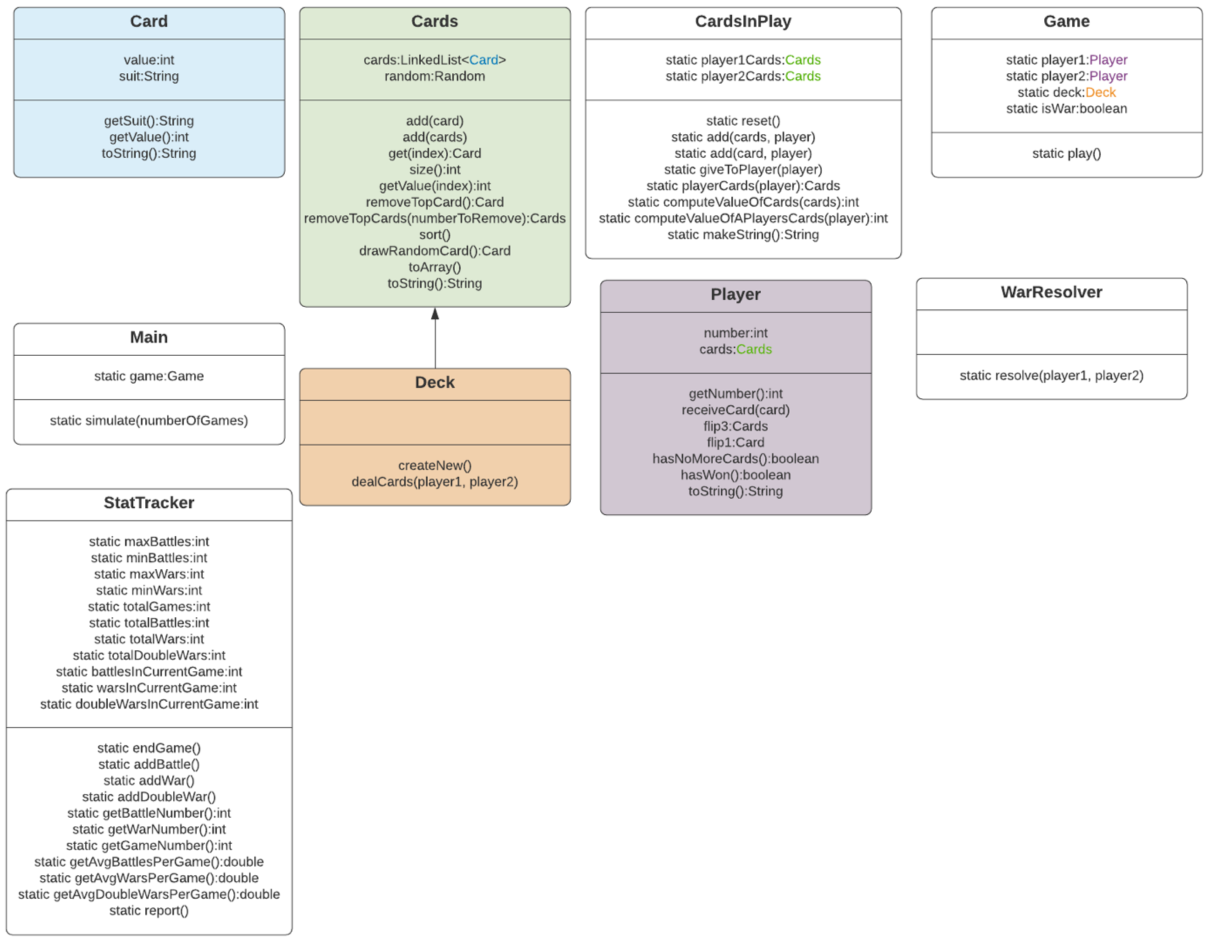
I recommend you design the following classes in implementing your solution. However, you are free to design however you prefer. This is ONLY a recommendation, and you don't have to follow it, nor will you be penalized for doing things your own way. You are also free to use the original/unrevised document I first sent you.

Before we get into details, let's list the classes I'm going to recommend.

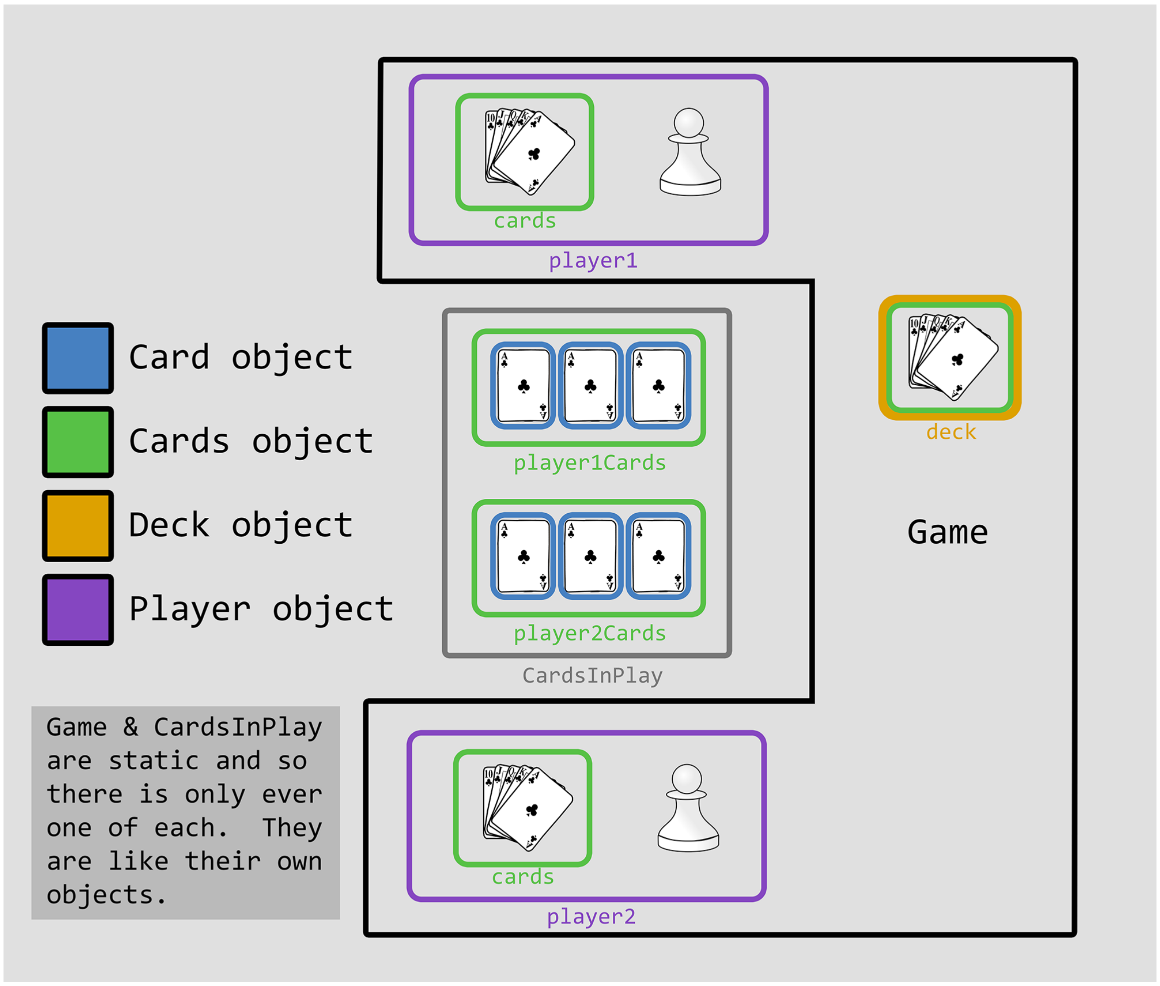
- **Main**: This will have your main method like usual. It's also going to use the Game class, where all the actual gameplay will happen. Here is where we're going to play a bunch of games by looping and using the Game class.
- **Game**: This class will have most of the rules to play a single game. It'll have the two players and the deck of cards. It'll handle all the game stuff EXCEPT wars.
- **WarResolver**: This is where all wars will be played out. This class is separate from game because the war code is complex and we don't want to clutter up the Game class.
- **Player**: The Player class handles all the stuff a player does. It has the player's hand of cards and can do things like play cards and receive cards and know if the player won.

- **CardsInPlay**: This class handles the cards that are in play during a round. These are the cards the players put on the table. It keeps track of each player's played cards and can do things like figure out the value of these cards, accept cards when someone plays them, and reward the winner with all the cards.
- **Cards**: Here we have a super useful class, which just represents a group of cards. This is used for the cards in the players' hands, the cards on the table (aka CardsInPlay), and for the deck. We can add a card to the group, get some cards off the top, draw a random card, and even sort them.
- **Deck**: A Deck is a special case of Cards. A deck can do everything a stack of cards can do, but more. A deck can deal itself to players, and a deck always starts with the 52 unique cards. Because of this, Deck is a subclass of Cards. Deck extends Cards.
- **Card**: This is the basic element of the game. A Card has a value and a suit. The Cards class is made up of a collection of this class. The deck is an extension of Cards, so it also uses this class. CardsInPlay also uses this class.
- **StatTracker**: This class does pretty much what it says – tracks the statistics in the game. We put all this into one class so we don't have it scattered around everywhere. This makes it easy to update and get our stats, since we just need to call methods on StatTracker.

Here's how the classes look in a UML diagram:



And here's what the composition looks like:



Note that some classes aren't on the composition image. That's because they aren't involved in composition. Like StatTracker is its own independent thing. It doesn't appear inside any other classes. Same with Main and WarResolver.

In OOP, it's important that each class is in charge of **one thing** and one thing only. When classes start being in charge of more than one thing, stuff starts to get messy. And confusing. And buggy. For this reason, I've got some classes that at first might seem extraneous. Why is the code for wars in a different class than the code for the game? Because wars are really a separate part of the game, and it's easier to isolate them in their own class. This is the same reason we have a StatTracker class – because Game and WarResolver and CardsInPlay shouldn't also be tracking stats. They have their own jobs. Same reason I didn't put Deck inside Game. The Game class should play the game. The Deck class should handle creating a deck and dealing the hands.

You should also strive for each method to do **one thing** and only one thing. There are many reasons to follow this practice: 1) Methods that do one thing are easy to test. As soon as a method starts doing more than one thing, it becomes hard and sometimes impossible to test. If you don't test your methods, you're going to have a bad time. 2) Methods that do one thing can be easily reused, incorporated in other methods, and chained together with other methods to do more complex things. 3) Methods that do one thing are inherently smaller and easier to understand, and readability is paramount when writing code. 4) Methods that do one thing are so much easier to write and revise.

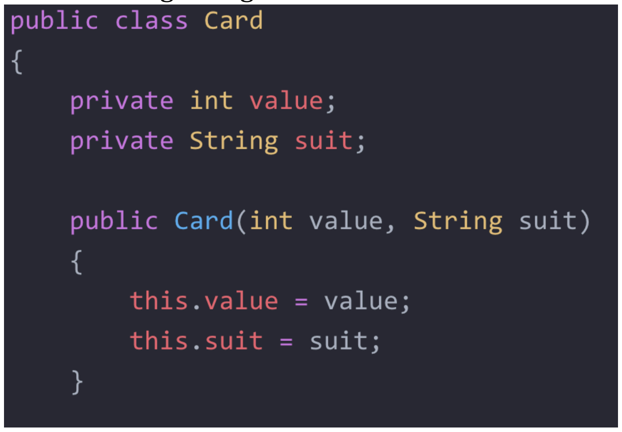
A note about how to proceed: I suggest starting off by writing the smaller classes and then testing them before moving on. First, make a Main class with just the main method. This way, you have a place to test things. Then I'd begin with Card. Write it, then in main try creating some cards and using the toString() method to see if stuff is working. Once you have Card down, I'd proceed in this order:

1. Card
2. Cards
3. Player
4. Deck
5. CardsInPlay
6. Game
7. WarResolver
8. StatTracker

Always test your classes and their methods using the toString() method to verify stuff is working. Only move to the next class once you're **sure** your class works. If you move on before this, you're going to have a bad time.

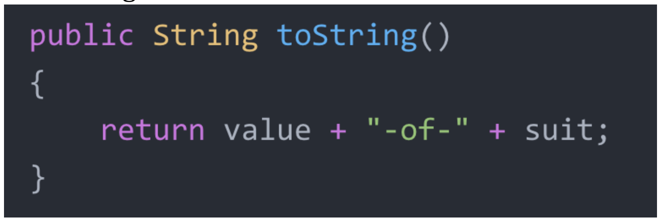
For example: here's how I'd go about writing Card.

1. Write the beginning of the class and the constructor:



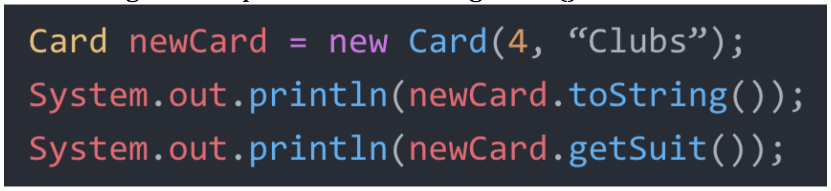
```java
public class Card
{
    private int value;
    private String suit;

    public Card(int value, String suit)
    {
        this.value = value;
        this.suit = suit;
    }
```

2. Write toString(). Always write toString() first so you can use it to test! Yours might looks something like this:



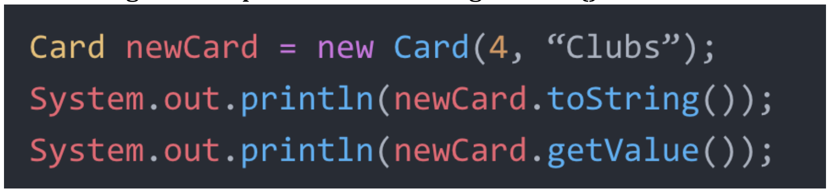
```java
public String toString()
{
    return value + "-of-" + suit;
}
```

3. Write getSuit(). In main, create a new card like so and call its toString() method to make sure it created right, then print the result of getSuit(), like this:



```
Card newCard = new Card(4, “Clubs”);
System.out.println(newCard.toString());
System.out.println(newCard.getSuit());
```

4. Write getValue(). In main, create a new card like so and call its toString() method to make sure it created right, then print the result of getValue(), like this:

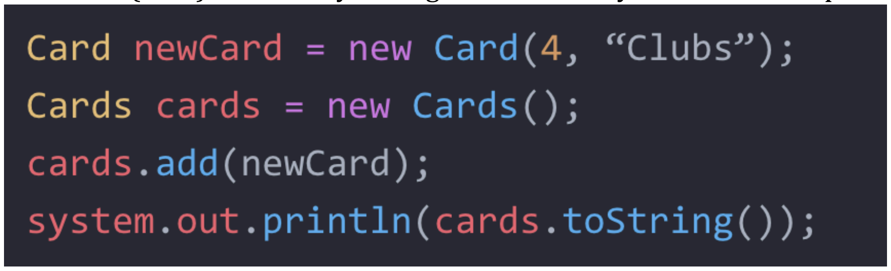


```
Card newCard = new Card(4, “Clubs”);
System.out.println(newCard.toString());
System.out.println(newCard.getValue());
```

Now we know that all the methods for Card work and we can confidently move on to the next class.

Here’s how I’d write one of the more complicated classes, Cards:

1. Write the beginning of the class and the constructor.
2. Write toString(). Always write this first so you can use it to test!
3. Write add(card) and test by calling it with a card you created and print toString(), like this:



```
Card newCard = new Card(4, “Clubs”);
Cards cards = new Cards();
cards.add(newCard);
system.out.println(cards.toString());
```

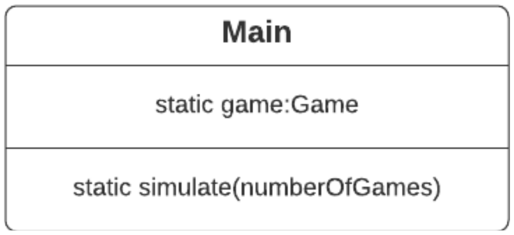
4. Write size(), add some cards, and print out the size()
5. Write add(cards) and test by making a Cards object, adding cards individually with add(card), then adding this other Cards object by calling add(cards) and printing toString()
6. Write get(index) and test by adding cards, calling it and printing the resulting Card’s toString()
7. Write getValue(index), add some cards, and print out the value of some cards
8. Write removeTopCard() and test by adding some cards, then removing the top card and printing the toString() for the Cards type it returns, then printing the toString() for the original Cards object to make sure the cards were removed.
9. Write removeTopCards(numberToRemove) and test by adding some cards, removing various numbers of cards and printing the toString() for the Cards type it returns, then printing the toString() for the original Cards object to make sure the cards were removed.
10. Write sort() and test by calling it and printing the result of toString() before and after the sort.
11. Write drawRandomCard(), add some cards, and test by making sure the cards it returns seem random (again, printing out the toString() for each card)

OK, you get the idea. But seriously, don’t come to me with issues unless you’ve tested your methods like this. This kind of thing is simple and makes a HUGE difference. **YOU SHOULD ALWAYS CODE LIKE THIS. Write toString() so you can use it for testing. Write a method. Test. Write the next method. Test. Etc.**

Note: In the class descriptions below, I've given you suggestions for member variables and methods you might want to use in the classes. Feel free to use other variables and methods! In fact, I've only suggested the public methods for a class (aka its **class interface** – the methods available to other classes). You are free to make lots of other methods (and should!). In fact, my code has lots of private methods that I didn't list here. However, you can get by just using the public methods I've listed…if you want to write big messy methods and not break them down into small, clean ones.

In the following description of the classes, I've included UML diagrams to help you visualize things. Static classes are white. Non-static ones are color-coded so you can see where they're used in other classes via composition. These are the classes I recommend:

Main
This class contains the main method for running the program. It'll receive the number of games to play via a command line argument (the String[] args thing that gets passed into the main method). Convert the string that's passed into args[0] into an int to find out how many games to play. (Google how to convert a String to an int in Java.)



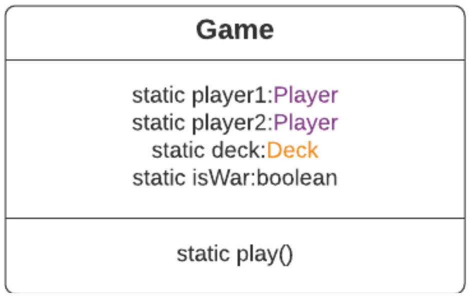
Member Variables:

- **game** *[static Game game]*
  This is the class that will actually play the game and implement the rules of War. It's static because we only have one game playing at any one time, so we don't need multiple instances of the Game class. We can just have one (aka static) game variable and use it.

Methods:

- **simulate** *[static void simulate(int numberOfGames)]*
  Static because we just need to run one simulate. Takes as a parameter the number of games to run. Simulate will then run the specified number of games. When done with all the games, it'll tell StatTracker (see below) to print out a final report.

Game
The game class is where the actual game gets played. Here's where players will flip three (or fewer) cards and see who wins. A game consists of two players, a deck of cards, and the cards that are currently in play (the ones out on the table). A method, *play*, carries out the rules of the game of war until one of the players has won. If needed, the play method can use WarResolver (see below) to resolve any wars. Everything in the Game class is static because we only ever have one game running at a time, so we never need to create multiple instances of Game.



Member Variables:

- **player1** *[static Player player1]*
  This will store all the info about player 1, including the cards in his/her hand (see the Player class below)
- **player2** *[static Player player2]*
  This will store all the info about player 2, including the cards in his/her hand (see the Player class below)
- **deck** *[static Deck deck]*
  The deck used to deal the cards at the beginning of the game. Once dealt, we don't need to use deck anymore, since it no longer has any cards in it (see the Deck class below)
- **isWar** *[static Boolean isWar]*
  Just a variable to set when a tie happens and we need to resolve the war (see WarResolver)

Methods:

- **play** *[static void play()]*
  Creates a couple players, creates a deck of cards, tells the deck to deal the cards to the players, and then plays the game until there's a winner. **When the game is done, calls StatTracker's endGame() method to record the end of the game.**

WarResolver

The WarResolver class does exactly what it says: it resolves the wars in the game. When the players put down cards and are tied, use this class to figure out who wins based on the wars that follow. WarResolver has static methods because we don't need to create multiple WarResolvers. We just need the one, so we make it static.

WarResolver
static resolve(player1, player2)

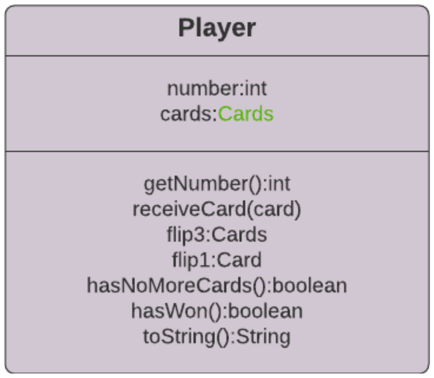
Methods:

- **resolve** *[static void resolve(Player player1, Player player2)]*
  To resolve a war, we call this method and pass in both players. We have the players flip a card each, compare, and figure out who won. If the war is a tie, we keep doing war until it's not a tie. If one of the players runs out of cards, the other player wins the war. If both players run out at the same time, it's a tie. If there's a winner, use CardsInPlay's giveToPlayer() method to give all the cards in

play to the player who won (see CardsInPlay below). At the end, we can call StatTracker's addWar() or addDoubleWar() to log that we had a war / double war (see StatTracker below).

Player

Here we define everything about a player: his/her player number (are they player 1 or player 2) and the cards in his/her hand.



Member Variables:

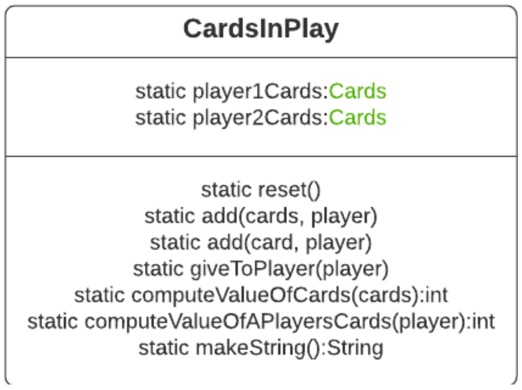
- **number** *[int number]*
  This just lets us know which player is which. When you create a player, you'll assign him/her a number (1 or 2). We can check it using the getNumber() method described below.
- **cards** *[Cards cards]*
  These are the cards in the player's hand. This isn't an array or a linkedlist. This is a type called Cards (see below) which will handle everything involved with a set of cards. The Cards class has the actual linkedlist in it. (It'll make more sense when we talk about the Cards class.) When we start a game, the Deck will deal cards to the players. So the cards will get removed from the Deck's cards and get put into this variable. When a player flips cards (puts them out on the table during a round of the game), the cards will get removed from this variable and put into the Game class's cardsInPlay variable.

Methods:

- **getNumber** *[int getNumber()]*
  Returns the player's number (1 or 2).
- **receiveCard** *[void receiveCard(Card card)]*
  When this gets called, we take the card passed into this method and add it to our cards variable (see Cards below)
- **flip3** *[Cards flip3()]*
  Removes three cards from our cards variable and returns them in the form of a Cards object
- **flip1** *[Card flip1()]*
  Removes one card from our cards variable and returns it in the form of a Card object
- **hasNoMoreCards** *[boolean hasNoMoreCards()]*
  True if the player has no more cards (i.e. the cards variable's size() method is 0)
- **hasWon** *[boolean hasWon()]*
  True if the player has all 52 cards (i.e. the cards variable's size() method is 52)

- **toString** *[String toString()]*
  Returns a string with the information about the player: the player's number and a toString() of the cards variable, which will list all the cards in the player's hand.

CardsInPlay
This class keeps track of the cards that are currently in play during a round of the game. These are the cards the players put down on the table as they try to win the round. So, when both players take 3 cards from their hand and put them on the table, the cards will go from the Player class's "cards" variable into the CardsInPlay's "player1Cards" or "player2Cards" variables. The variables and methods in this class are all static because we only ever have one set of cards in play. We don't need to instantiate multiple copies of this, so it's static.


CardsInPlay
static player1Cards:Cards
static player2Cards:Cards
static reset()
static add(cards, player)
static add(card, player)
static giveToPlayer(player)
static computeValueOfCards(cards):int
static computeValueOfAPlayersCards(player):int
static makeString():String

Member Variables:
- **player1Cards** *[static Cards player1Cards]*
  These are the cards in play that belong to player 1.
- **player2Cards** *[static Cards player2Cards]*
  These are the cards in play that belong to player 2.

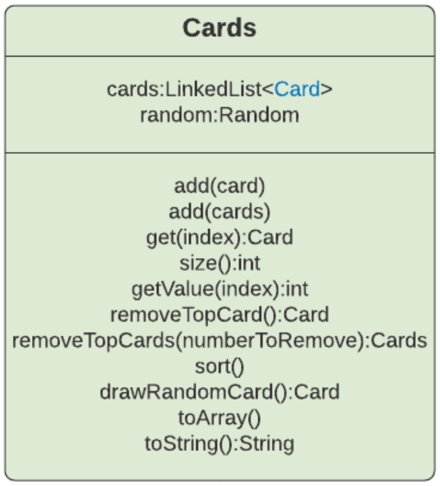
Methods:
- **reset** *[static void reset()]*
  This empties the player1Cards and player2Cards variables so there are no cards in play. (We just set the variables to new Cards() like this: player1Cards = new Cards()). **We call this at the beginning of each game because we don't want to start a game with any cards in play.**
- **add** *[static void add(Cards newCards, Player player)]*
  This method takes a Cards type (a group of cards) and adds it to the player1Cards or player2Cards variables. Like if a player flips 3 cards, we'd call this with the flipped cards and the player, use player.getNumber() to figure out the player's number, and then add the cards to either player1Cards or player2Cards.
- **add** *[static void add(Card newCard, Player player)]*
  This method takes a Card type (a single card) and adds it to the player1Cards or player2Cards variables. Like if a player flips 1 card, we'd call this with the flipped card and the player, use player.getNumber() to figure out the player's number, and then add the card to either player1Cards or player2Cards.
- **giveToPlayer** *[static void giveToPlayer(Player player)]*
  Gives all the cards in play to the player that was passed into the method. So if we passed in player1,

this would give every card in player1Cards and player2Cards to player1. You'd go through each card in player1Cards and then each card in player2Cards and use the player's receiveCard() method to give the card to the player. **When done, use reset() to clear out all the cards in play (since they were all given to a player).**

- **computeValueOfCards** *[static int computeValueOfCards(Cards cards)]*
  Goes through all the cards passed in via the cards variable and adds up their values (see Card below and its getValue() method). Returns this sum of their values. This can be used when creating a toString() or debugging.
- **computeValueOfAPlayersCards** *[static int computeValueOfAPlayersCards (Player player)]*
  Goes through all the cards for the player passed in (either player1Cards or player2Cards – you can figure out which player it is by using the player's getNumber() method) and adds up their values (see Card below and its getValue() method). You can use computeValueOfCards() above to make this easier by passing in the cards of a player. Returns this sum of their values. Useful in Game to figure out who won a round.
- **makeString** *[static String makeString()]*
  Since everything in this class is static, we can't use the normal toString() method. (This is because we're overriding an existing toString() method in Object, which if you'll recall all Java classes inherit from. That toString() method isn't static, so we can't override it with a static toString() method.) So we just name it something different. I chose makeString(), but you can use any name for the method. Returns a string with the information about the cards in play: player 1's cards in play and player 2's cards in play. You can use the toString() method in Cards (see below).

Cards

A class that represents a group of cards. Two cards. Fifty-two cards. Any number of cards. This can be used to make a deck, to keep track of the cards in a player's hand, or represent the cards in play on the table during a round. It's just a class that lets us store multiple cards and do stuff with them.



NOTE: You'll need to import some stuff in order to use LinkedLists, Random, and to do the Comparator for sorting. **Make sure you have these at the top of your class file**:

```
import java.util.Comparator;
import java.util.LinkedList;
```

import java.util.Random;

Member Variables:

- **cards** *[LinkedList<Card> cards]*
  A linked list of Card types. This is the list that stores each card. Note that each thing in the list is a Card type (aka the Card class you made earlier).
- **random** *[Random random]*
  An instance of the Random class. We'll use this when we need to produce a random card from our cards linked list. Don't forget to create the instance in your constructor! (random = new Random()).

Methods:

- **add** *[void add(Card card)]*
  Adds the single card to our linked list cards.
- **size** *[int size()]*
  Returns an int telling how many cards are in our linked list. You can use the linked list's own size() method to find this out.
- **add** *[void add(Card cards)]*
  Adds more than one card to our linked list cards. Note that the variable passed in is also of type Cards (aka the class you're currently making!). Weeeeeeird.
- **get** *[Card get(int index)]*
  We give this method an index, and it'll find the card at that position in the linked list (you can use the linked list's own get() method) and returns that card. Doesn't remove the card from our list.
- **getValue** *[int getValue(int index)]*
  We give this method an index, and it'll find the card at that position in the linked list (you can use the linked list's own get() method), then returns the value of the card. (You can call Card's own getValue() method!)
- **removeTopCard** *[Card removeTopCard()]*
  This method removes one card from the beginning of the linked list and returns it. To remove something from a linked list, use the remove() method and pass in the index of the element you want to remove. So to remove the first thing, you'd call remove(0). The remove method will also return back what it removes, so you can do this: return cards.remove(0);
- **removeTopCards** *[Cards removeTopCards(int numberToRemove)]*
  This method removes some number of cards from the beginning of the linked list, puts them into a new Cards object, and returns that object. Before you do anything, make sure you check to see there are enough cards to remove! If you only have 2 cards and try to remove 3…bad time. So maybe if *numberToRemove* is more than the number of cards, set *numberToRemove* equal to the number of cards. Make a new Cards object to temporarily store the cards you're removing. Then loop through the cards linked list for a numberToRemove times, remove the first item from the linked list (you can use the removeTopCard() method you just wrote!), and put it into the temporary new Cards object. When you're done, return that temporary new Cards object.
- **drawRandomCard** *[Card drawRandomCard()]*
  Picks a random card and returns it. To do this, pick a random index for the cards linked list by doing random.nextInt(cards.size()). That nextInt() method will pick a number between 0 and one less than the number you pass in. So if we have 10 cards, cards.size() will be 10, and the indexes for it will be 0 to 9. So nextInt(10) will give us numbers from 0 to 9. Once you have a random index, you can use the remove() method on cards to remove the item at the random index and return it. As a bonus, here's what the code could look like:

```java
protected Card drawRandomCard()
{
    int randomCardIndex = random.nextInt(cards.size());
    return cards.remove(randomCardIndex);
}
```

- <u>**sort** *[void sort()]*</u>
  Sorts the cards by value. Because sorting involves the use of a comparator, which we haven't learned about, I'm going to give you the entire method right here:

```java
public void sort()
{
  cards.sort(new Comparator<Card>()
  {
      @Override
      public int compare(Card card1, Card card2)
      {
          if (card1.getValue() < card2.getValue())
              return -1;
          if (card1.getValue() > card2.getValue())
              return 1;
          return 0;
      }
  });
}
```

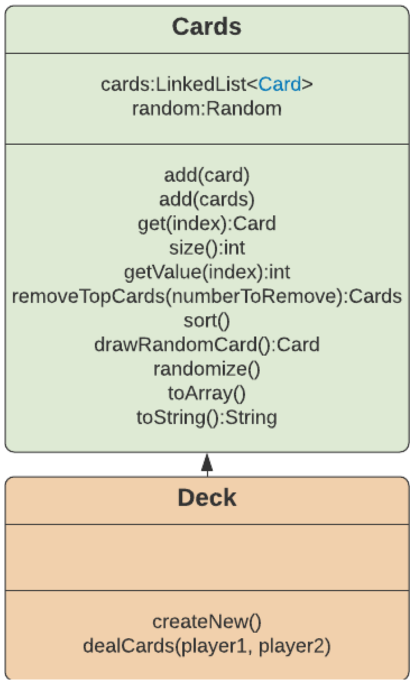
- <u>**toArray** *[Card[] toArray()]*</u>
  This method will convert the linked list to an array and return the array. YOU WILL NEED THIS FOR THE WARLOGGER CODE. Here's some more bonus code:

```java
public Card[] toArray()
{
    Card[] cardArray = new Card[cards.size()];
    for (int i = 0; i < cards.size(); i++)
    {
        cardArray[i] = cards.get(i);
    }

    return cardArray;
}
```

- <u>**toString** *[String toString()]*</u>
  Creates and returns a string with all the info for the cards. You could format it with commas separating each card or whatever. Use the toString() method in Card (see below).

Deck

The Deck class extends the Cards class. This means it inherits from Cards. Remember how Dog was a subclass of Pet? Deck is a subclass of Cards. Deck has all the stuff in Cards, plus its own stuff. Basically, Deck is a specific type of Cards. It's Cards but with the ability to deal to players. Decks also start with all 52 cards.



Member Variables: all the same as Cards, since Deck inherits from Cards. You don't have to put these variables in Deck, though. You already put them in Cards, and Deck can just magically use them as if you'd also typed them into the Deck class. NOTE: for Deck to use the variables in Cards, you have to make them public or protected in Cards. If you make them private in Cards, Deck won't be able to see them!

Methods: all the same as Cards, since Deck inherits from Cards. You don't have to put these methods in Deck, though. You already put them in Cards, and Deck can just magically use them as if you'd also typed them into the Deck class. NOTE: for Deck to use the methods in Cards, you have to make them public or protected in Cards. If you make them private in Cards, Deck won't be able to see them!

Deck also has a couple methods that are unique to it. These are what make a deck a Deck and not just a Cards.

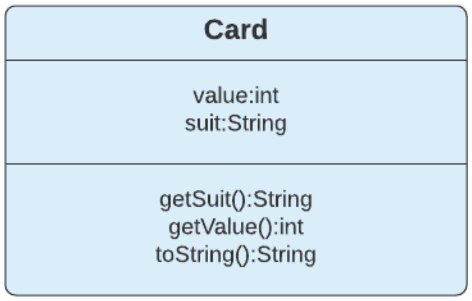
- **createNew** *[void createNew()]*
  This is where you're going to make 52 cards and add them to the cards linked list (the one defined in Cards that Deck can use because it inherits from Cards). You're going to create a new Card and set its value and suit (see Card below). Then add the card to cards. If you don't want to have to type in 52 lines of code to make 52 cards, you can use loops! Try this:

```java
String[] suits = new String[] {"Heart", "Spade", "Club", "Diamond"};
for (String suit : suits)
{
  for (int value = 2; value <= 14; value++)
    {
      add(new Card(value, suit));
    }
}
```

- **dealCards** *[void dealCards(Player player1, Player player2)]*
  Goes through the cards in the cards linked list. Gives one to player1, then one to player2 (you can use Player's receiveCard() method). You can use the drawRandomCard() method that Deck inherits from Cards to make this super easy. Because I'm getting tired of typing explanations, check out this bonus code:

```java
public void dealCards(Player player1, Player player2)
{
    while (cards.size() > 0)
    {
        player1.receiveCard(drawRandomCard());
        player2.receiveCard(drawRandomCard());
    }
}
```

Card
A card consists of a value (an int that represents 2-10, J, Q, K, A in their integer form – so 2-14, where Jack is 11, Queen is 12, King is 13, Ace is 14) and a suit (a string for hearts, spaces, clubs, diamonds).



Member Variables:
- **value** *[int value]*
  The number value for the card. So a Jack would be 11, a Queen would be 12, a King would be 13, and an Ace would be 14.

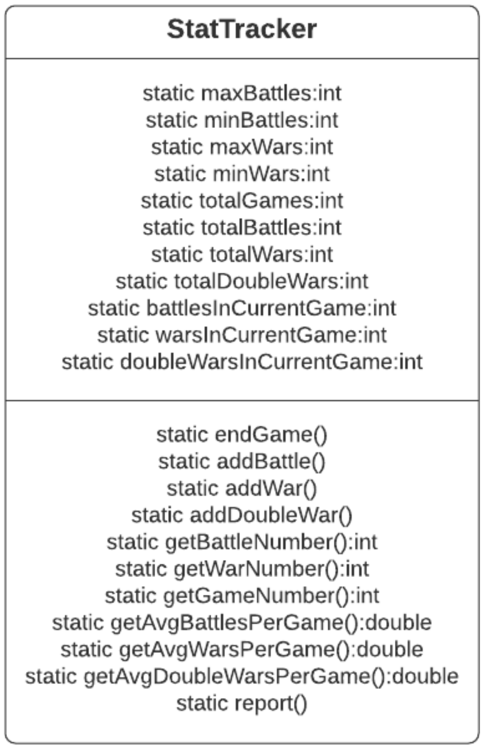
- **Suit** *[String suit]*
  A string representing the suit ("Diamonds", "Clubs", "Hearts", "Spades")

Methods:

- **getSuit** *[String getSuit()]*
  Returns the suit.
- ***getValue*** *[int getValue()]*
  Returns the value.
- **toString** *[String toString()]*
  Creates and returns a string representation of the card ("2 of Hearts" or "14 of Clubs") or if you're in a fancy mood ("Two of Hearts" or "Ace of Clubs")

StatTracker

This method does what you would expect: it tracks all the statistics for the games. Because we only need one StatTracker and don't need to create multiple objects of it, everything in it is static. It contains a lot of member variables and methods, but it's actually quite simple.


StatTracker
static maxBattles:int
static minBattles:int
static maxWars:int
static minWars:int
static totalGames:int
static totalBattles:int
static totalWars:int
static totalDoubleWars:int
static battlesInCurrentGame:int
static warsInCurrentGame:int
static doubleWarsInCurrentGame:int
static endGame()
static addBattle()
static addWar()
static addDoubleWar()
static getBattleNumber():int
static getWarNumber():int
static getGameNumber():int
static getAvgBattlesPerGame():double
static getAvgWarsPerGame():double
static getAvgDoubleWarsPerGame():double
static report()

Member Variables:

- **maxBattles** *[static int maxBattles]*
  Stores the largest number of battles across all the games played. So if one game had 518 battles and that was more than any other game, maxBattles would be 518.
- **minBattles** *[static int minBattles]*
  Stores the smallest number of battles across all the games played. So if one game only had 13 battles and that was fewer than any other game, minBattles would be 13.
- **maxWars** *[static int maxWars]*
  Stores the largest number of wars across all the games played. If one game had 11 wars and that

was more than any other game, maxWars would be 11. Note that this also includes double wars or triple or whatever. As long as a war occurs (no matter how long it goes on), it counts as one war.

- **minWars** *[static int minWars]*
  Stores the smallest number of wars across all the games played. So if one game had 0 wars, minWars would be 0.
- **totalGames** *[static int totalGames]*
  Stores the number of games played.
- **totalBattles** *[static int totalBattles]*
  Stores the number of battles across all games.
- **totalWars** *[static int totalWars]*
  Stores the number of wars across all games.
- **totalDoubleWars** *[static int totalDoubleWars]*
  Stores the number of double wars across all games. A double war is when a war starts and the players each put down a card, but the cards are a tie. A new war starts to break the tie. This is called a double war. NOTE: if the double war is a tie and proceeds to another tiebreaker (a triple war, etc), these do NOT count as double wars. Only record double wars. Not triple or quadruple or etc.
- **battlesInCurrentGame** *[static int battlesInCurrentGame]*
  Used to store the battle count for the current game
- **warsInCurrentGame** *[static int warsInCurrentGame]*
  Used to store the war count for the current game
- **doubleWarsInCurrentGame** *[static int doubleWarsInCurrentGame]*
  Used to store the double war count for the current game

Methods:

- **endGame** *[static void endgame()]*
  Called when a game ends. This will update all the stats (add the battlesInCurrentGame to the totalBattles etc), update the min/max stats, increase the totalGames, and finally reset all the stats that only apply to a single game (battlesInCurrentGame, warsInCurrentGame, doubleWarsInCurrentGame).
- ***addBattle*** *[static void addBattle()]*
  Called after a battle happens. Increases battlesInCurrentGame.
- ***addWar*** *[static void* addWar*()]*
  Called after a war happens. Increases warsInCurrentGame.
- ***addDoubleWar*** *[static void* addDoubleWar*()]*
  Called after a double war happens. Increases doubleWarsInCurrentGame.
- ***getBattleNumber*** *[static int* getBattleNumber*()]*
  Returns the number of battles so far in the current game (battlesInCurrentGame). **You need this for WarLogger.**
- ***getWarNumber*** *[static int* getWarNumber*()]*
  Returns the number of wars so far in the current game (warsInCurrentGame). **You need this for WarLogger.**
- ***getGameNumber*** *[static int* getGameNumber*()]*
  Returns the number of games so far (totalGames). **You need this for WarLogger.**
- ***getAvgBattlesPerGame*** *[static double* getAvgBattlesPerGame*()]*
  Divides totalBattles by totalGames and returns the result
- ***getAvgWarsPerGame*** *[static double* getAvgWarsPerGame*()]*
  Divides totalWars by totalGames and returns the result

- ***getAvgDoubleWarsPerGame*** *[static double* getAvgDoubleWarsPerGame*()]*
  Divides totalDoubleWars by totalGames and returns the result
- **report** *[static void repor ()]*
  Prints out a report with the following information:
  Average number of battles per game
  Average number of wars per game
  Average number of double wars per game
  Max number of battles in a game
  Min number of battles in a game
  Max number of wars in a game
  Min number of wars in a game

WarLogger

The WarLogger class is provided for you and is used to log the internal details of your game for grading purposes, but other than calling its methods you need not be concerned with it. **DO NOT EDIT THIS CODE FOR ANY REASON. YOU MUST USE THIS CLASS! DO NOT WRITE YOUR OWN!** You should use the WarLogger for the following:

After a Battle is Dealt
After a battle is dealt (cards given to both players), you should call for each player: WarLogger.getInstance().logBattle(…) passing in the correct parameters. Note there are constants defined in the WarLogger class to represent player 1 and player 2. For example, to log the hand dealt for battle number 10 for player 2, you would call:
WarLogger.getInstance().logBattle(10,WarLogger.P2,h) where h is an array of cards held by player 2 for the battle.

NOTE: this method wants an array, but our cards are stored in a linked list. Use the toArray() method we wrote in the Cards class to convert our cards to an array.

I call it like this (AFTER I've called StatTracker.addBattle()):

```
Cards player1Cards = CardsInPlay.playerCards(player1);
Cards player2Cards = CardsInPlay.playerCards(player2);
WarLogger.getInstance().logBattle(StatTracker.getBattleNumber(), WarLogger.P1,
  player1Cards.toArray());
WarLogger.getInstance().logBattle(StatTracker.getBattleNumber(), WarLogger.P2,
  player2Cards.toArray());
```

After a Battle is Won
To record the outcome of a battle, call WarLogger.getInstance().logBattleOutcome(…) passing in the correct parameters.

Make sure you call StatTracker.addBattle() first. Note that winnerForWarLogger is a string that I set to WarLogger.P1 or WarLogger.P2 or WarLogger.WAR if neither player won and the battle went to a war. Like, literally it looks like this:

```
String winnerForWarLogger = WarLogger.P1;
```

I call logBattleOutcome() like this:

```
WarLogger.getInstance().logBattleOutcome(StatTracker.getBattleNumber(), winnerForWarlogger);
```

After a War is Won
To record the outcome of a war, call WarLogger.getInstance().logWarOutcome(…) passing in the correct parameters.

Make sure you call StatTracker.addWar() first. I call logWarOutcome() like this (where winnerForWarLogger is a string that I set to WarLogger.P1 or WarLogger.P2 depending on who won the war):

```
WarLogger.getInstance().logWarOutcome(StatTracker.getWarNumber(), winnerForWarlogger);
```

After a Game is Won
To record the outcome of a game call WarLogger.getInstance().logGameOutcome(…) passing in the correct parameters.

Make sure you call StatTracker.endGame() first. I call logGameOutcome() like this (where winnerForWarLogger is a string that I set to WarLogger.P1 or WarLogger.P2 depending on who won the game):

```
WarLogger.getInstance().logGameOutcome(StatTracker.getGameNumber(), winnerForWarLogger);
```

At the End of Your Simulation
THE LAST LINE OF YOUR MAIN METHOD IN THE SIMULATION CLASS SHOULD BE:
WarLogger.getInstance().release();

NOTE: It's possible to get into an infinite loop if the cards happen to always be collected in the same order. As an easy fix, when a player collects cards and adds to the bottom of the stack, you can add them in random order. However, it's still possible to have bad luck and still infinitely loop…. One other thing you can do is check StatTracker's getBattleNumber() method, and if it's really large (like over 1000 battles), that's a sign the game can't end. You could then end the game and start over without increasing the totalGames count so you'd basically be replaying the game again.

Being OO
It is expected that your class implementations adhere to OO best practices. Methods should be given the correct access specification (public or private) depending on their purpose and use. Appropriate constructors should be provided for each class. Accessor/Mutator methods should be provided when there is a need to return/alter member variables. You DO NOT need to write constructors or accessor/mutator methods if they're not used. Don't just write getters/setters for everything.

You are of course free to add as many classes/methods as you like to get the job done, but keep in mind that you should strive for an elegant, efficient solution.

Due Date

This assignment is due at 11:59 pm on 5-4-2020. Submit the appropriate zip file via email. It should be named LastNameFirstInitial_MP5. Please make sure to include all the required files (README, source files).

Grading

Assignments will be graded on correctness, adherence to style, and the inclusion of meaningful comments where needed.