Jeffrey Ma
jma@caltech.edu

Console-Based, Offline Multiplayer Thirteen

Program Requirements:

1) A working installation of Java 11 (has not been tested on other versions)
2) A working installation of the GNU make utility
3) A working installation of git (optional)

Instructions for starting the game:

1) Clone or download the files from the following link and navigate to your downloaded directory using the `cd` command in your UNIX shell. This can be done with either `git clone` or through downloading through a browser.

   https://github.com/18jeffreyma/thirteen-console.git

2) I have provided a Makefile for this Java project to simplify building and compiling. As long as a working installation of Java 11 is present on your system, the project should compile with no issues. Type in the following to your command line:

   ```
   make
   ```

3) This will compile the downloaded `.java` files into `.class` files. To run the game, you can type in either of the following options:

   `make run {name1} {name2} ...` or `java ThirteenGame {name1} {name2} ...`

   The arguments that follow should be the names of each of the players who will be playing as well as the order that you will be playing in. A starting player will be chosen with the game rules, but the order will remain the same for the entire game.

4) From here on, the program will provide instructions on how to play, so enjoy and have fun! This is a offline multiplayer game, where all players will play on a single device.

About Thirteen (Game Rules):

Thirteen is a card game invented in Vietnam and Southern China, in which the objective of the game is to get rid of all of your cards. In a standard four-person game (though the game can be played with more or less), each player is dealt 13 cards, hence the origin of the name.

The game can be explained in several parts, which are listed below:

The Ranks of the Cards:

For easy memory, in several other regions, a variation of Thirteen known as “Big Two” is played, which describes the ranking of the cards from *lowest to highest*, which are:

3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace, 2

Additionally, suits also have a rank in this game, and from *lowest to highest*, they are:

Spades ♠, Clubs ♣, Diamonds ♦, and Hearts ♥

Note that the 3 of spades is the lowest single card in the game, while the 2 of hearts is the highest ranked. Thus, the rules of the game state that the player who has the 3 of Spades has the starting round of the game (this program will automatically start with that player). If we play with less than four players, and the 3 of Spades is not dealt, the game will default to the player with the lowest ranked card.

The Five Types of Plays:

In Thirteen, there are five types of plays:

1) Singles, where you play a single card (i.e. 8♠).
2) Pair/Doubles, where you play two cards of the same value (i.e. 9♣, 9♦)
3) Triples, where you play three cards card of the same value (i.e. 3♠, 3♣, 3♥)
4) Straights, where you play three or more cards that are in a sequence (6♠, 7♣, 8♥, 9♣)
5) Bombs, where you play four cards of the same value (i.e. A♠, A♣, A♥, A♦)

For the first four listed above, note that if the player before you plays a single, you must also follow with a single (the same for doubles, triples, etc). If you wish to play a straight, you must have the same length and have a larger straight than the player before you. The special case is for bombs, which can be played when any other play is on the pile. In the event a bomb is played on a single, pair, triple, or straight, in order to "beat" the bomb and win the pile, you must have a bomb of a higher number. An implementation note, for this program, suits only contribute to the size of a card when singles are in play (i.e. 3 ♠ is larger than a 3 ♣)

We'll talk about later what it means to win the pile.

General Gameplay Flow

The general game flow is as follows:

1) Deck is shuffled and cards are dealt (either 13 cards per person for <4 players or an even number of cards per person for 5+). The player who was dealt the 3 of Spades or the lowest card in play will go first.

2) Turns will proceed in a sequential fashion based on a predetermined order.

3) When it is your turn, you can decide if you wish to Play or Pass.

    a. If you wish to Pass, you are unable to play again until all other players in the round have passed (i.e. no jumping in). Note that you must pass if you are unable to make a valid play.

    b. In order to make a Play, you must play a hand that is "higher" than the hand currently on the top of the pile. Also, the hand you play must be of the same type as currently on the

pile (e.g. if the first player played pairs, everyone must play pairs until the round is over). The notable exception to this is playing a bomb, which changes the type of the pile to bombs only.

4) A player wins the pile when all other places pass, either by choice or by inability to make a move. When this happens, the pile is cleared, and this player can start the pile with whatever play they wish.

5) The winner is the first player to get rid of all of their cards, at which point the game will end.

Design

Language Choice

For my choice of language, I picked Java, which I believe to be a great language for this kind of problem. Games are inherently object-oriented, and so picking Java allowed me to implement this game in a way that both logically follows the real world in terms of object structure and is readable from a developer standpoint. Additionally, the Java standard library contains several already implemented data structures which I took advantage of in my project.

Libraries Used

```
java.util.List;
java.util.ArrayList;
java.util.LinkedList
```

- Used for backing data structures for Thirteen game pieces.

```
java.util.Comparator
java.util.Collections;
```

- Used for defining and sorting an ordering of cards.

```
java.lang.object.StringBuilder
```

- Used heavily for `toStrings()` because repeatedly adding Strings is very inefficient (Strings are immutable in Java).

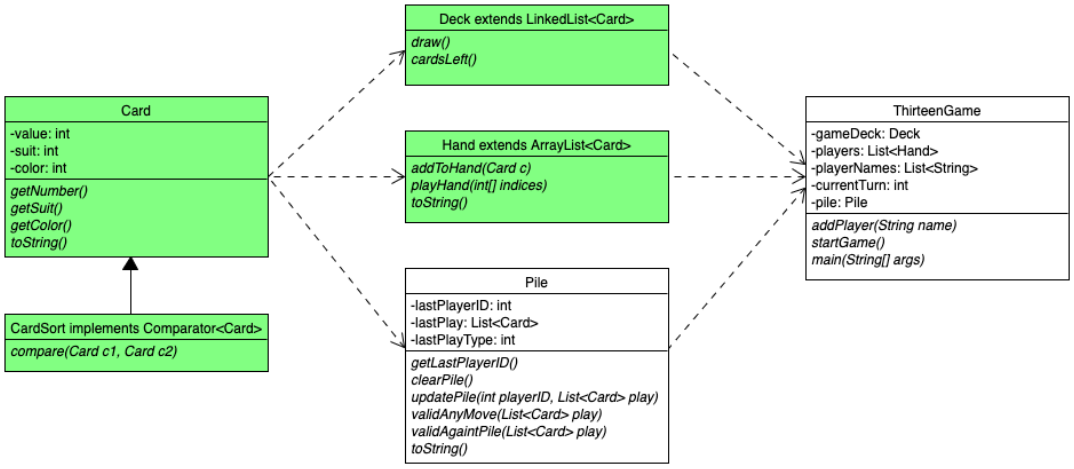
```
java.util.Scanner
```

- Used to get input from the console for player moves.

```
java.lang.Exception
```

- Used for input sanitization and making sure the user is entering valid information in try-catch blocks. Used `IOException, IllegalArgumentException,` and `NumberFormatException.`

Basic Game Pieces



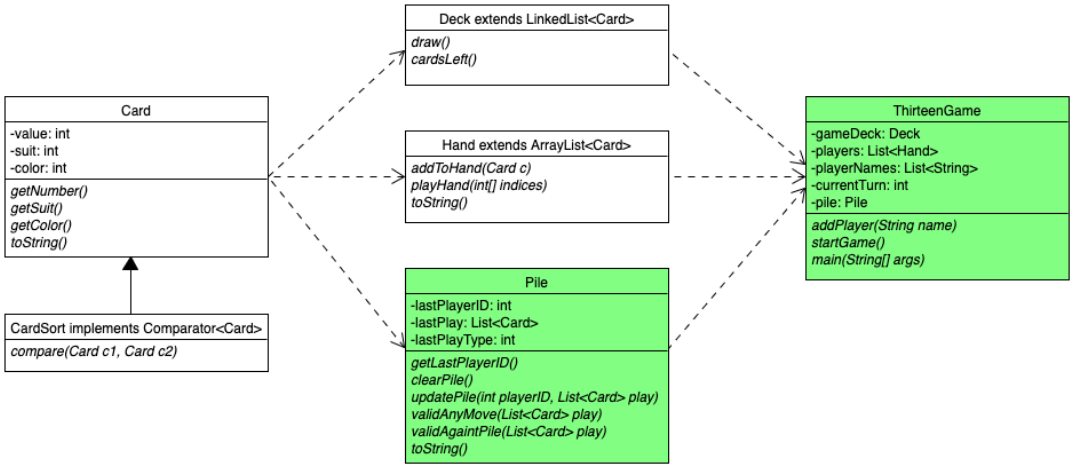
I approached developing this game from the bottom up, first starting with the game pieces themselves. Since Thirteen is a very comparison-based game, I built my `Card` class with integers representing the suits and values, to make comparison much simpler. In addition, since I was tasked with building a console based game, overriding `toString()` for the `Card` made printing in higher abstractions much simpler. Finally, to make code more readable, for non-numerical card values such as Jack, Queen, Kings, etc. as well as for suits, I used static final integers in my `Constants` class to assign cards to those values. If you're more curious about this, feel free to take a peek into the `Constants.java` file.

Additionally, since I knew I wanted to take advantage of Java's built in `Collections.sort()` for lists method, I implemented the `Comparator` interface within my Card class for easy comparison later on.

Moving to the `Deck` class, I saw a `LinkedList` as the best way to back this game piece. Like in the real world, without opening a deck, we can only see the cards at end of the deck. Furthermore, most of a deck's work is done during shuffling, so we would prefer if drawing a single card was as efficient as possible (which is `O(1)` in a linked list). While not really necessary for a game such as Thirteen, where players do not draw from the deck once the game starts, I thought implementing it this way most logically followed its real-world counterpart. This class only really contains one key method, `draw()`, which removes a card from the top of the Deck.

From there, I implemented the `Hand` class, backed by an `ArrayList` and chosen because during a game, we want efficient access to any card inside our hand. This class implementation also closely mirrors how a person would play in real life: for example, the `addtoHand()` method is a sorted insertion into the `ArrayList`, similar to how in real life we would add a dealt card to our hand in some kind of sorted manner. The other notable method is the `playHand()` method, which given a set of indices, returns a list of Cards (this is called by an outside process when the player makes a play). The overriding of `toString()` in the `Card` class makes our overriding of `toString()` for printing a `Hand` object much simpler here, as we only need to add a small bit of formatting.

Specific Gameplay Elements



Moving on to specific gameplay elements, a key part of Thirteen is determining whether a play is valid or not, which usually involves comparing a player's input play against the play currently on top of the middle pile. Thus, I implemented a `Pile` class which keeps track of the last play played onto the pile, the type of play it was, as well as who it was played by. Keeping track of the type of play allows us to dictate what the next player must play, and keeping track of who it was played by allows us to determine if a player has won the pile (if everyone has passed and it's his/her turn again). A `Pile` object also validates given plays and signals to the game process if a play is accepted or not.

Thus, I implemented several methods, the most notable being `updatePile()` and `validAgainstPile()`. The former method receives a `List<Card>` from the `playHand()` method in `Hand`, and uses the later method to check if a submitted move is valid and updates the pile if it is so. If not, the method returns false, indicating to whatever process that called it that an invalid move occurred. Additionally, I also overrode `toString()` for the `Pile` class to print pile information in a readable format.

Finally, the last class I implemented was the main class `ThirteenGame`, which contains a `Deck`, a list of `Hands` and a list of `Strings` representing players and their names, an integer indicating whose turn it is, and a `Pile` object. Upon starting the game with the main method, we enter into a while-loop which we will refer to as the game loop. When a player begins their turn, they enter into another infinite while loop, known as the player loop. This allows us to catch if the player is making invalid non-number inputs or invalid game moves and continue looping until the player has made a valid move, after which the program will break from the player loop and move on to the next player. The game loop is only broken when a player runs out of cards, signaling a win.