Final Project Report

Author: Tianyi Lu (Sky)

Trust Game

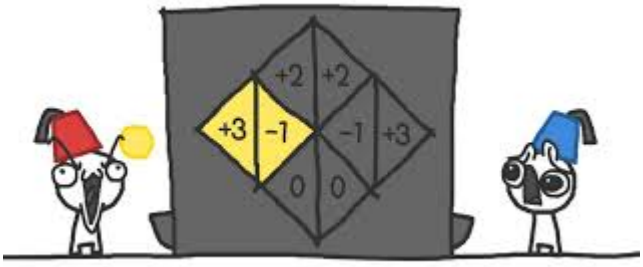
This project is modified on a game called "The Evolution of Trust" made by Nicky Case.
The original version can be played here: https://ncase.me/trust/

Prerequisite

- Any operating system and console except `Windows Powershell`.
  - `Windows powershell` has trouble displaying colorful outputs.
- Console to run this program must has width greater than `100 px`.
  - Some ASCII art may not display properly if running on smaller consoles.
- `JVM` installed

Why it's interesting and substantive?

The basic component of `Trust Game` is a simple match. In one match, two players are standing on the opposite sides of a machine. For each step, every player can choose to whether or not to put a coin in the machine. The number of coins each of them will gain or lost according to their action are shown as follows:



If we record the decision history of ourselves and the opponents, we can create simple AIs (or `bots`) that use these informations to make its own decisions based on certain strategies.

We can either play with these `bots` to figure out how to maximize our coins, or let different `bots` play against each other to see which strategy gets more coins.

`Trust Game` has **three** different game mode and **six** kind of bots to play with. It utilizes class contents such as `Inheritance`, `Sorting`, `HashMap` and `Stack`, and also implements advanced contents such as `Functional Interface`, `Method reference`, `Stream operations` and `ASCII art ouput`.

How to run the project?

First, you should complete everything by typing the following command in the console:

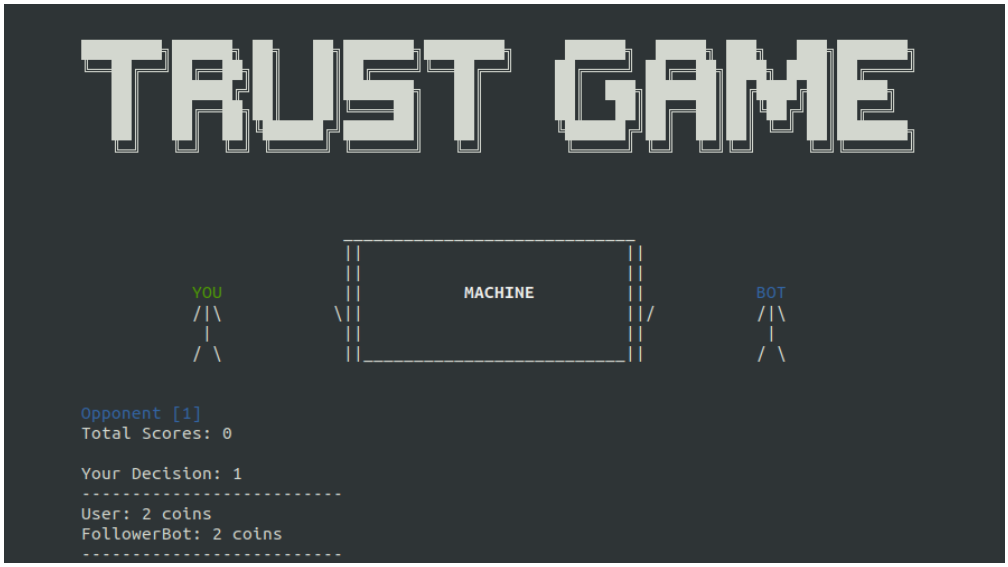
```
javac *.java
```

You can start with the full `StoryMode` by typing:

```
java Main
```

The output you get will be something similar to this:

The `StoryMode` will start from scratch and let you familiarize with the game step by step. Then, as the story proceed, you will enter `CompeteMode` :



In this mode, you will play against five different bots with their own strategies. You goal is to maximize the number of coins you can get. The strategies of each bot are as follows:

```
* FollowerBot
Hello! I start with Cooperate, and afterwards,
I just copy whatever you did in the last round.

* MeanBot
I always Cheat. The strong shall eat the weak.

* NiceBot
I always Cooperate. Let's be friends!

* GamblerBot
Listen. I'll start Cooperatin', and keep Cooperatin',
but if y'all ever cheat me, I'LL CHEAT YOU BACK 'TIL
THE END OF THE WORLD!

* DetectiveBot
First: I analyze you. I start: Cooperate, Cheat,
Cooperate, Cooperate. If you cheat back, I'll act
like FollowerBot. If you never cheat back, I'll act
like MeanBot, to exploit you. Elementary, my dear Watson.
```

Finally, you will reach `ArenaMode`. In this mode bots will play against each other. After each round the players with the lowest scores will be eliminated and the players with the hightest scores will reproduce to fill in the gap. Each player will be represented using `*` with their color in a circle:

You can play the `ArenaMode` by guessing the possible winner.

If you want to try `CompeteMode` and `ArenaMode` independently, you can type:

```
java Main -a
```

for `ArenaMode` and

```
java Main -c
```

for `CompeteMode`

Superclass and Subclasses

All the classes end with `Bot` implement `Bot interface`. The interface defines essential methods that every bot must realize. `Bot interface` also extends `Comparable interface` to enable sorting.
The `FollowerBot` is the Superclass for all the other bots:

- FollowerBot
  - MeanBot
  - NiceBot
  - GamblerBot
  - DetectiveBot
  - AlterBot
  - UserBot

`Inheritance` was especially useful in this program because every bot has the same instance variables and methods except `move` method (their unique strategies) and `constructor`. After realizing all methods in `FollowerBot`, we only need to write these two methods for other bots to make the program more concise and well structrued.

The other convenience `inheritance` and `interface` provides is that we can use `Bot` as the unified type to create `List`, `Map` and other data structure.

Other Class Methods

HashMap

```
private Map<String, Integer> settings; //TrustGame.java line 19
private Map<String, Integer> pSettings; //TrustGame.java line 20
private Map<String, BotConstructor> botMap; //BotMap.java line 21
```

HashMap is used in `TrustGame.java` to store the key value pair of game settings. I want a way to store different settings' name and value together. HashMap's nature of quick store and access key value pairs make it the best choice.

Sorting

```
Collections.sort(players); //TrustGame.java line 283
```

Sort algorithm is used to sort all the players(bots) by their score after each match. Because I want to find the 10 players with the higest and lowest scores, sort came in handy.

Other

Functional Interface & Method Reference

Using `String` as key and `constructor reference` as value can also enable me to create any bot I like just by using:

```
Bot newBot = botMap.getBot(botName); //TrustGame.java line 288
```

```
//BotMap.java Line 11
@FunctionalInterface
interface BotConstructor {
    public Bot construct();
}
...
//BotMap.java Line 23
public BotMap() {
    botMap = new HashMap<String, BotConstructor>();

    botMap.put("FollowerBot", FollowerBot::new);
    ...
}
```