Project 1: Showdown
Due: 5:00PM, February-23-2018

Contents

1 Let’s Play Cards! 1

2 The Story 1

3 The Game (from 1000 feet) 1
   3.1 Card Types 2
   3.2 Game Play 2

4 The Game (More Details) 2
   4.1 Advancing Status 3
   4.2 Show Traits 3
   4.3 Advantages and Disadvantages 3
   4.4 More Things to Consider 4

5 The Game (Still More Details) 4
   5.1 First, Some Definitions 4
   5.2 The Heroes 4
   5.3 The Control Cards 5
   5.4 More Specifics 5

6 Demo 5

7 Assignment 6
   7.1 Command-Line Arguments 6
   7.2 Deck Creation 6
   7.3 User Interaction 7
   7.4 Parsing Tips 7
   7.5 Output Logs 8
   7.6 Game event 9
      7.6.1 Prompt 10
      7.6.2 Warning 10
1 Let’s Play Cards!

For this project, you’ll be implementing a text-based card game. In this game, two players partake in a showdown. The rules of the game are somewhat complex, but they are fully explained in this handout. This project is intended to teach you about object-oriented program design. So we want you to spend the bulk of your time on just that—design—not puzzling over the rules. Please do not hesitate to ask the TA staff as many questions as necessary about the game rules.

In theory, it would be possible to complete this project using functional programming. However, the goal of this project is to practice using an object-oriented approach to tackle a large programming problem. The object-oriented perspective facilitates code reuse, rather than repetition, and showcases the relationships between various entities in a program.

OOP also facilitates program extensibility. Indeed, to test the merits of your object-oriented design, and in particular its extensibility, after design checks, we will release a list of additional cards that you will be required to implement in your final handin.

2 The Story

You have been summoned to help heroes from across the world fight each other so that they can train to become stronger and smarter when the ultimate battle comes! Unfortunately, you are too busy studying to really be able to meet these heroes in person, so you have come up with an idea. You will use a simulated card game to remotely manage and advise these heroes on their moves. You will be able to decide when they attack, defend, call for help and, finally, win the day!
To make the stakes higher, you have decided to make this training session a tournament, where the hero who is strong enough to battle until no other heroes can win!

3 The Game (from 1000 feet)

Showdown is a two-player game. Each player has the following:

- A deck. Each deck consists of two types of cards: Hero cards and Control cards.
- A hand. A player begins the game with six cards in their hand. There is no limit to the number of cards a player may eventually hold in their hand.
- A field. Each field consists of the bench and the arena. There can be up to six Heroes on each player’s bench at a given time, but each player can only have one Hero in their arena.
- A discard pile. Upon leaving the field, Heroes are sent to the discard pile for resting during the training.

Play alternates between the two players until one player wins by attacking the other when they do not have a Hero card in their arena.

3.1 Card Types

As already mentioned, there are two types of cards: Hero cards and Control cards.

Hero cards have a name, a team, a max HP (Health Points), and an attack, the latter of which is characterized by a damage value. A Hero may only attack if they are in the arena, and they may only attack another Hero (i.e., if the opponent also has a Hero in the arena). When they do attack, they inflict damage on the opposing Hero. Specifically, the damage of their attack is subtracted from the opposing Hero’s HP. When a Hero’s HP falls to zero or below, they have lost all strength to continue battling and are sent to the discard pile.

Control cards are played on Heroes on the current player’s field. Playing a Control card carries out its effect, as detailed in Section 5.3. Once its effect is completed, the Control card is removed from the game and cannot be reused.

3.2 Game Play

At the start of their turn, a player draws a card from their deck and adds it to their hand. If the deck is empty, no card is drawn.

The player whose turn it is then performs an unlimited number of actions. These may include:

- Moving a Hero from their hand onto the field. If the arena is empty, the new Hero fills it. Otherwise, the new Hero goes to the next available bench spot.
- Moving a Hero from the arena onto the bench, a Hero from the bench to the arena, or switching the Hero in the arena with one on the bench.
• Using a Control card from their hand.

To end their turn, a player can either attack with a Hero or pass. If a player does not have a Hero in their arena, then they may not attack. Because attacking signifies the end of a turn, players can only attack once per turn.

Obviously, Heroes cannot attack on the turn that they are played. Otherwise, the first player to put a Hero into the arena would win immediately.

4 The Game (More Details)

There are just a few more details to this card game. We describe them all here.

4.1 Advancing Status

When a Hero is first played, it is played in its lowest-level. But as play progresses, Heroes are able to advance their level to become stronger and more powerful. You can advance any Hero (on the field or on the bench) to a higher status (in other words, level-up) by playing a particular Control card (see Section 5.3).

A turn is defined as one complete turnover. For example, say you place a Hero on the field during your turn. At the beginning of your next turn, that Hero has been on the field for 1 turn. At the beginning of your next turn, it has been there for 2 turns, and so on. If the Control card is played on a Hero that has been on the field for 3 or more turns, that Hero can advance to its next form.

The following rules apply to all Heroes as they advance their status.

• When a Hero advances its status, its max HP generally goes up, but any damage inflicted remains. So if Hero had 40 HP (out of a max HP of 60) before advancing in status, and the new status should have afforded it max HP 100, your new advanced Hero will have only 80 out of a max HP of 100.

• Heroes only “know” one attack at a time; they do not “learn” new attacks. Once a Hero is status-advanced, it has a new attack and “forgets” any previous attacks.

• When a Hero advances its status, any alterations it has (such as an increased/decreased HP or an attack) and any traits it has (see Section 5.3) are lost. Only damage carries over.

• When a Hero advances its status, it must be on the field for 3 turns before it can advance again.

4.2 Show Traits

The lowest-level forms of some Heroes enter the field with specific traits, which are forgotten whenever Heroes advance status.

• A Hero with the Defenders of Justice trait (i.e., a Hero in the Justice League) must attack (unless, of course, it entered the field that turn).
• A player with a Hero with the *Healers of the World* trait (i.e., a Hero in *The Avengers*) can gain 5 HP (up to its maximum HP) at the end of the player’s turn if it is in the arena but did not attack.

### 4.3 Advantages and Disadvantages

Since the Heroes come from different universes, it makes sense that some would have natural advantages and disadvantages over the others, and indeed they do.

Specifically, if Hero $A$ has a team advantage over Hero $B$, then $A$’s attack will do 20 more damage than usual to $B$. Also, $B$’s attack will do 20 less damage than usual to $A$. An attack can never do negative damage.

Note that advantages and disadvantages are always mirrored. This means that if team $X$ has an advantage over team $Y$, then team $Y$ is at a disadvantage compared to team $X$.

### 4.4 More Things to Consider

• Moving a Hero to the bench from the arena or vice versa does not alter its HP.

• A Hero may never have a HP higher than its max HP. However, do note that max HP can be increased using a Control Card (more on that later).

• If you play a card at any time and its effect cannot be performed, the card is simply discarded.

  Examples:
  - You play a Hero from your hand, your arena is occupied, and you already have six Heroes on the bench.
  - You play a Control card and there is no Hero in the field position selected.

• The only way to remove a Hero from the field is for it to lose all HP and be sent to the discard pile.

• If a player attempts to attack, but does not have a Hero in the arena, their turn still ends. Likewise, if a player attempts to pass, but there is a Hero in the arena who must attack (e.g., *Defenders of Justice*), the player attacks.

### 5 The Game (Still More Details)

In order to maintain consistency, we have defined a set of cards that you must implement. You may implement other cards for fun, but the project you turn in should support all of the required cards.

#### 5.1 First, Some Definitions

• **The Avengers:** A team of Earth’s mightiest heroes appearing in Marvel Comics universe

• **Justice League:** A team of powerful superheroes in DC Comics universe, striving to fight crime and injustice
5.2 The Heroes

There are two different teams: The Avengers and Justice League.

<table>
<thead>
<tr>
<th>The Avengers</th>
<th>Justice League</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Panther</td>
<td>Wonder Woman</td>
</tr>
<tr>
<td>Iron Man</td>
<td>Cyborg</td>
</tr>
</tbody>
</table>

These Heroes are characterized by their status measured in levels, their max HP, and their attacks (with a damage value), all of which are listed below:

<table>
<thead>
<tr>
<th>Hero</th>
<th>Status</th>
<th>max HP</th>
<th>Attack</th>
<th>Damage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Panther</td>
<td>Level 1</td>
<td>60</td>
<td>Curved Claw</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Level 2</td>
<td>100</td>
<td>Panther Bite</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Level 3</td>
<td>120</td>
<td>Sudden Death</td>
<td>60</td>
</tr>
<tr>
<td>Iron Man</td>
<td>Level 2</td>
<td>100</td>
<td>Unibeam</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Level 3</td>
<td>150</td>
<td>Smart Bomb</td>
<td>50</td>
</tr>
<tr>
<td>Wonder Woman</td>
<td>Level 1</td>
<td>60</td>
<td>Shield Smash</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Level 2</td>
<td>100</td>
<td>Lasso of Truth</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Level 3</td>
<td>140</td>
<td>Amalthea Bash</td>
<td>50</td>
</tr>
<tr>
<td>Cyborg</td>
<td>Level 2</td>
<td>40</td>
<td>Power Fist</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Level 3</td>
<td>70</td>
<td>Sonic Disruptor</td>
<td>30</td>
</tr>
</tbody>
</table>

5.3 The Control Cards

- **HP Regain**: Choose a Hero. It regains all lost HP.

- **HP Boost**: There are two types of HP Boost: The Avengers, and Justice League. Any HP Boost may be played on any Hero. If their team matches the card, the Hero regains 30 HP. If their team does not match, that Hero regains only 10 HP (they are still able to regain HP).

- **Max HP Boost**: Choose a Hero. The max HP of that Hero increases by 20. It also gains 20 HP.

- **Rampage**: Draw three cards and add them to your hand.

- **Level Up**: Choose a Hero to advance to its next level if it can (see Section 4.1 for details).

5.4 More Specifics

Here is the system of advantages/disadvantages:

- Justice League Heroes have an advantage over The Avengers Heroes.

And here are the traits:

- The Avengers Heroes have the trait *Healers of the World*.

- Justice League Heroes have the trait *Defenders of Justice.*
6 Demo

We have provided you with a fully functional version of the Showdown cards game. If you have any questions about how the game is played, or about user interaction, your first resource should be the demo. You can run it directly in the terminal using the following command:

```
showdown_demo <Player1 Name> <Player1 Deck> <Player2 Name> <Player2 Deck>
```

A fully functioning implementation should match our demo exactly, so due note when certain outputs are printed (i.e., a single input can trigger multiple events, so be sure to keep track of all of these).

You will find example deck files in `/course/cs0180/src/showdown/`. Upon release, the only deck file is `DeckPreExtension.deck`. To run the demo, assuming you have copied the deck file into your working directory, use the following command:

```
showdown_demo Mae DeckPreExtension.deck Alex DeckPreExtension.deck
```

You should be able to guess from the name of this file that another deck is forthcoming, namely `DeckPostExtension.deck`.

7 Assignment

You are responsible for writing a program that reads a user’s input, performs the game action corresponding to that input, and outputs a message explaining the outcome of that action. This behavior should repeat until the game ends.

7.1 Command-Line Arguments

Your program for this project should expect four command-line arguments. The first and third arguments should be the names of the two players, while the second and fourth arguments should be the paths to the players’ respective deck files.

To run your program from the command line in the directory `~/course/cs018/workspace/javaproject/bin/`, use the command:

```
java showdown.sol.Game <Player1 Name> <Player1 Deck> <Player2 Name> <Player2 Deck>
```

**Your project’s main method must live in the class** `Game`.

7.2 Deck Creation

The first thing your program needs to do is construct each player’s deck. To make this task easier, we have provided a `ShowIO` class (`/course/cs018/src/showdown/src/ShowIO.java`) with a static method `getDeck`. This method takes as input a `String` containing the path to a `.deck` file. This file should store a deck, constructed as a list of cards, one per line, like this:
Black Panther
Rampage
Level Up
Wonder Woman

The `getDeck` method is intended to read this file, and return a `List` of `Card`s (which it will do once you have created a `Card` class or interface). Each card in your list should correspond to one line of the deck file. You can then use this card list to create a deck.

**Note:** If you include the word “shuffle” _anywhere_ in the file, the order of the cards will be randomized.

### 7.3 User Interaction

The `ShowIO` class also includes a static method `getNextInput` that reads a line of input from the user. After the players’ decks are constructed, your program should use this method to prompt the user for input, and then it should perform the appropriate action. This process should repeat until the game is over.

There are six valid commands that players can issue, the syntax of which they must follow exactly. If the user inputs an invalid command, you should log an error message and prompt the user for another input. Invalid input should _not_ cause your program to crash!

The commands a player can enter are as follows:

- **print field**: Prints all Heroes in both players’ fields, and their HPs.
- **print hand**: Prints your hand, and the number of cards left in your deck.
  **Note:** Players should be able to input `print field` and `print hand` at any time.
- **attack**: The Hero in your arena position attacks, ending your turn.
- **switch (number)**: Switches your arena position Hero with the Hero in the `<number>` spot on the bench. If that spot is empty, the Hero in the arena position just goes to that spot on the bench.

  Positive integers are nicer for indexing than natural numbers, when writing a user-facing program (or game). Consequently, your `switch (number)` command should assume that the bench is _1-indexed_—that is, the first Hero sitting on your bench is considered to be at position 1. The Hero that is in the arena is considered to be at position 0. Likewise, your `play (number)` command should also be _1-indexed_.

- **play (number)**: Chooses a card from your hand to play. If it is a Hero card, it is put on the field (if there is no room in the arena or on the bench, it is discarded). If it is a Control card, its effect is carried out.

  **Note:** Since playing a Control card often involves choosing which Hero it should affect, you may need to accept a second input from the user (in addition to the `play (number)` command). You should _not_ create your own I/O objects to do this; rather, you should make use of what is provided in the `ShowIO` class.

- **pass**: Ends your turn.
7.4 Parsing Tips

Java has a few built-in methods that you may find useful in parsing user input. For example, the `String` class provides a method called `split`, which is used to divide a `String` into separate words:

```java
java> String example = new String("I am an example")
java> String[] myArray = example.split(" ")
myArray = {"I", "am", "an", "example"}
```

The `Integer` class provides a static method called `parseInt` for converting a `String` into an `Integer`. For example:

```java
java> String number = "18"
java> int myInt = Integer.parseInt(number)
myInt = 18
```

Be careful with `Integer.parseInt`; if your `String` input isn't a valid integer, it will throw a `NumberFormatException`. However, this Exception should not reach the user! Instead, you should `catch` the Exception using a `try-catch` block. For instance, if in the following piece of code a `NumberFormatException` is thrown in the `try` block, the content of the `catch` block will be executed instead:

```java
try {
    int myInt = Integer.parseInt("Not a number!");
} catch (NumberFormatException nfe) {
    System.out.println("You just tried to input something that's not a number!");
}
```

7.5 Output Logs

Whenever a player requests some action, it is certainly very important that your project correctly performs that action. However, if the player is not informed that the action was properly carried out, what good is it that the action was performed? Because of this, we have very particular requirements for what and when you must output to the player (using the given `Logger` class).

We have provided you with classes that correspond to the various types of output you will have to handle. There are four categories: `GameEvent`, `Prompt`, `GameWarning`, and `InputError`. Each method in the `Logger` class takes in one of these classes (or, that is, a class that extends one of these classes) as input. These will be explained in detailed in Section 7.6.

All necessary classes are provided in the `/course/cs018/src/showdown/src/` directory. All outputting must be done with the `Logger`—that is, do **not** ever use `System.out.println()`. This is very necessary for how we will be testing your project.

**NOTE:** Trying to understand how to use logger or how the event classes structure works will be confusing if you do not look at the source code first. **Every given (non-abstract) event class should be used at some point throughout the game, so make sure you keep track of when you need to log**
output. Please take sometime to familiarize yourself with the source code, and as always, if you have any questions, please come to TA hours or ask on Piazza!

7.6 Game event

A GameEvent should be logged whenever a valid action occurs while the game is in progress. For example, when a Hero is attacked by another Hero, a HeroAttacked event should be logged (the HeroAttacked class, in turn, has a pre-written toString() method that the Logger outputs to the user). Note, however, that if that Hero was just played, this should instead log a warning, as a Hero is not allowed to attack on the turn it enters the field. The GameEvents (and their expected outputs) are as follows. Each game event corresponds to one call to Logger.logGameEvent.

Using the logger:
When logging a GameEvent, simply entering a String message will not suffice. GameEvent is a class (see source code for more details), so say for example that you are playing the card Rampage (see CardPlay in src code). Then you can log this event by:

```java
Logger.logGameEvent(new CardPlay("Rampage"));
```
<table>
<thead>
<tr>
<th>Game Event</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting the game</td>
<td>new GameStartEvent(String p1, String p2)</td>
</tr>
<tr>
<td>Player successfully draws a card (including at the beginning of the game)</td>
<td>new CardDraw(String p1, boolean deckNotEmpty)</td>
</tr>
</tbody>
</table>
| **print field** | new PrintFieldEvent(CardDescription[] cards, String player)  
new PrintFieldEvent(CardDescription[] cards, String player)  
Note this should be called twice; first for the current player, and then for the other player. See the CardDescription constructor for how to instantiate this. |
| **print hand** | new PrintHandEvent(String player, List<String> cardNames, int cardsRemaining) |
| Player uses switch | new PrintFieldEvent(CardDescription[] cards) |
| After a card has been played (before prompting for more input) | new CardPlay(String cardPlay) |
| Hero evolves | new HeroEvolve(String cardName, String evolvedCardName) |
| Player uses a health card | new HPBoostPlay(String cardName, int currHP) |
| Player uses HP Regain | new HPRegainPlay(String cardName, String totalHP) |
| Hero is attacked | new HeroAttacked(String currPlayer, String currPlayerCardName, String otherPlayer, String otherPlayerCardName) |
| An attack succeeds (i.e., this comes directly after the above log) | new SuccessfulAttack(String attackedCardName, int currHP, int totalHP) |
| Hero is out of HP | new NoHP() |
| Game receives an EoF | new EOF() |
| **Defenders of Justice** trait activates | new DefendersOfJusticeActivate(String cardName) |
| Player attacks an empty arena | new EmptyArenaAttack(String currPlayer) |
| The game ends | new GameOver(String winningPlayer) |

### 7.6.1 Prompt

A **Prompt** should be logged whenever the game prompts a player for input. Some things that should be logged as prompts include:

- When the game prompts you to select a target for a Control card
- When the game prints the options a player has

#### Using the logger:

When logging a game prompt, similar to GameEvent, you will have to use a Prompt object. For instance, when asking the player **Mae** to make a move, simply log the following:

```
Logger.logGamePrompt(new MovePrompt("Mae"));
```
where player is the player whose turn it is to make a move. We also provide a ChooseHeroPrompt class for when a player needs to select a target for a Control Card.

7.6.2 Warning

A GameWarning should be logged when a player makes a move which isn’t legal according to the rules of the game. For example, if a player were to try to play a Hero when their bench and arena were full, you would output a warning log, since there is nowhere to put that Hero. The content of warnings is up to you, but please be informative (or simply follow the demo). Other Warnings include:

- When a player attempts to evolve a fully evolved Hero
- When a player attempts to play a Control card on a location where no Hero is
- When a player tries to attack when their arena is empty
- When a player tries to attack on their Hero’s first turn on the field

Using the logger:

```java
Logger.logGameWarning(new GameWarning("You cannot attack on the turn you enter the arena"));
```

7.6.3 Error

Finally, an error should be logged when the player uses the program wrong. Unlike warnings, which were generated by valid inputs that violated the game rules, errors are for inputs that cannot be interpreted to have any effect in the game. For example, a player who tries to access the 7th card in their hand when they only have 3 cards should see an error log. Other errors include:

- When a player inputs a non-numeric input when prompted for a number
- When a player inputs an invalid command
- When a player refers to an invalid location on the field

Logger:

```java
Logger.logError(new InputError("Invalid location"));
```

Check out the demo (see Section 6) to get an idea of what your game should look like. The functionality of your program should be very similar to that of the demo. In general, your grader’s happiness will increase with the degree to which your handin is similar to the demo, and it’s certainly in your best interest to make your grader happy!
8 Testing

Thoroughly testing your project will involve both unit testing and system testing.

8.1 Unit Testing

Unit tests attempt to establish the correctness of small bits of functionality within a program, assuming everything else works as intended. While all code should, in theory, be unit tested, for this project, you need only carry out a few unit tests. Some good pieces of functionality to unit test include:

- Attack damage – with respect to team advantages and disadvantages
- Healing – with respect to team advantages and disadvantages
- Evolutions – with respect to current HP, max HP, attack damage, etc.

Because these are unit tests, you need not carry out a full action in the game to test functionality (e.g., you don’t necessarily need to have two Heroes attack each other to test attack damage). It is perfectly legitimate to unit test a class using only that class’ methods.

8.2 System Testing

System testing attempts to establish the correct functionality of an entire system, in which multiple chunks of code interact with one another. In this project, thorough system testing will require you to play your game repeatedly, invoking multiple different behaviors, and making sure that all methods work as intended.

To demonstrate to us that you performed thorough system tests, you should run through multiple games, and record and submit your inputs and the resulting outputs. Play all cards, special powers, type advantages, etc., and record the resulting effects. Then try invalid commands, indices, etc., and record the error messages received. Add this documentation to the end of your README file in a section entitled Testing. Include a header for each of your tests that state what functionality is being tested.

To make this testing process more efficient, you can create a text file of commands that can be run through your programs using a terminal trick known as “redirection” (see Section 8.3). You can create one (or preferably several) of these files to test the functionality of your game, with each one leading to a different game outcome. If you would like to split your write up into multiple files, feel free; however, please describe where and what those files are in your README so your grader knows what you did.

Your testing documentation should be sufficient to convince yourselves (and your TAs) that you have thoroughly tested your program.

8.3 Redirection

Redirection is the process of changing where functions like readLine, println, and other I/O operations read or write text. When you run a program, every time it calls readLine, it waits for
the user to type some input, and every time it calls println, it outputs some text to the terminal. Not so with redirection. With redirection you can specify where readLine reads lines from, where println outputs text to, and much more! Redirection is accomplished with two meta Heroes, ‘<’ and ‘>’, which redirect input and output, respectively.

Rather than playing your Showdown program interactively by entering commands one after another (which could get tedious after hours of debugging), you can create a file listing a series of commands to be executed (in order), say commands.txt, and then run your program on that file. To do this, you would redirect that file to your program, so that functions like readLine would treat the file commands.txt as standard input, instead of waiting for input from the usual standard input (the keyboard). Here is how to specify that your Showdown program should redirect its standard input to commands.txt:

```
java showdown.sol.Game Mae testDeck.deck Alex testDeck.deck < commands.txt
```

Redirecting output is very similar; simply add ‘> output.txt’ to the end of the previous command. This will write everything that would have been printed to standard output to the specified file (creating it if it does not exist, and overwriting it if it does).

One final note about redirection: When the end of a text file is reached, readLine (and getNextInput, which calls it) reads ‘EOF’, and returns null. This may cause your program to crash with an ol’ time favorite—a NullPointerException. To avoid this fate, you should test for null wherever an ‘EOF’ could be read. While testing, you can simulate an ‘EOF’ from the command line by entering ‘Ctrl-d’ when prompted for input. In addition, the ShowIO class contains a method eofReached that simply returns whether or not ‘EOF’ has been reached. Feel free to use it if you’d like.

### 9 Extensible Design

Computer programming is an iterative process. You design a program, and then you write a program; and then you try to extend it, and you realize you need to re-design it—repeatedly. Consequently, during the design process, you should already be thinking about how your programs could be extended if desired. For this card game, for example, you can think about how you would extend your program if:

- new Heroes and Teams were added
- new Control cards were added
- Heroes gained new special effects, activated either on attack or at some other point during a turn

To test the extensibility of your design, after design checks, we will release a list of additional cards that you will be required to implement in your final handin. Don’t worry; we won’t change any rules or fundamentals of the game; but you should be sure to keep extensibility in mind as you work on the core of your program’s design.

To implement some of required functionality of this project, such as calculating team advantages and disadvantages, you might find that you it would helpful if you knew the types of the objects
you were dealing with. Keep in mind that you are not allowed to use `instanceof`, or any related methods, to ascertain the type of an object. This is generally considered bad programming practice and does not lend itself to extensibility. If you run into this problem, we encourage you to spend time re-thinking your design. In addition to visiting the TAs on hours for guidance, see Appendix A, which describes a clever workaround that lends itself to extensibility.

You are, however, allowed to use `instanceof` in your `equals` methods (only). **Note:** In this project (only), you are not required to implement `toString` and `equals` methods for every class, but only where you find them useful.

## 10 Handin

### 10.1 Design Check

Design checks will be held on 2–13–18 and 2–14–18. We will send out an email detailing how to sign up for design checks; it is in your best interest to sign up as soon as possible (before all the prime time slots have been filled).

**Reminder:** You are required to pair program at least the design check portion of all CS 18 projects. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check without one.

For the design check, you must do the following:

- Draw a class diagram showing all the interfaces and classes you plan to implement, and how they are related. You should also indicate the members each will have, and what purpose they will serve. You need not have written any code yet.

- Know each step of a turn, and be able to tell us what will happen after each of the possible valid user inputs. In other words, show us that you know which methods your program will execute in response to user input, and where those methods will be defined.

- Be sure to read the **Double Dispatch** section, as this will be very important in your implementation, and ask the TA any questions you have.

### 10.2 Final Handin

The final handin is due by 2–23–2018. For the final handin, your `javaproject` should contain the packages `showdown.src` (in the `src` directory) and `showdown.sol` (in the `sol` directory). Your code should be part of the `showdown.sol` package. That package should also contain a `README.txt` file.

Your `README` file should include:

- instructions for use, describing how a user would interact with your program

- a brief overview of your design, including how all of the pieces of your program fit together

- a description of any features you failed to implement, as well as any extra features you chose to implement
- a description of any possible known bugs in your program
- a description of how you tested your program
- a list of the people with whom you collaborated
- a testing section, with appropriate headers, inputs and outputs for each system test

To hand in your files, navigate to the `~/course/cs018/workspace/javaproject` directory, and run the command `cs018_handin showdown`. This will automatically hand in the contents of your entire `javaproject` directory. Once you have handed in your project, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

Note: Only one of you or your partner must hand in the project.

Note: Please note that it is an academic code violation to post your code in a publically viewable way on the internet. If you are considering using GitHub to facilitate collaboration between you and a partner, please make sure that your code is in a private repository. You can create a student account on GitHub that gives you an unlimited number of private repositories for free. If you don’t know what GitHub is, don’t worry about it! It is far from necessary to use GitHub for your CS 18 projects.

10.3 Grading

As with all CS 17/18 projects, a good design will make coding this project significantly easier; so you should spend a fair amount of time working on your program’s design before you begin writing any code.

The design check counts for 15% of your grade, including:

- Your class hierarchy and class diagram. Is your design suitably extensible?
- A walkthrough of your planned responses to various user inputs.

The goal of this project is for you to familiarize yourself with, and then apply, basic principles of object-oriented design. As such, 15% of your grade will assess your design only.

Functionality counts for 50% of your grade, including:

- Correct implementation of basic game mechanics.
- Correct implementation of all required cards.
- Correct application of user inputs.

As always, partial functionality merits partial credit.

The final 20% of your grade will be reserved for comments, testing, and style. You should include documentation comments and test cases for all non-trivial methods. You should also perform system testing, to test interactions among methods. Additionally, comment any code which would otherwise be unclear.
A Double Dispatch

Searching desperately for ways to practice your new found object-oriented programming skills, you decide to model the behaviors of your four favorite head TAs—Alex, Mae, Eleanor, Liam—in Java. Knowing how special each individual one of them is, you decide to define an abstract class `Person`, and then subclass that class for each head TA. Here's what your `Person` class looks like:

```java
package doubleDispatch;

/**
 * An abstract class that represents persons.
 */
public abstract class Person {

    String name;
    String greeting;

    /**
     * A constructor for Person
     */
    public Person() {
        this.name = "Person";
        this.greeting = "Hey There! I'm a Person!";
    }

    /**
     * Talks to another person by printing a greeting.
     * @param aPerson - the person to talk to
     */
    public void talkTo(Person aPerson) {
        System.out.print(this.greeting + " ");
    }

    /**
     * Specific behavior for talking to Alex
     */
    public abstract void talkTo(Alex aAlex);

    /**
     * Specific behavior for talking to Eleanor
     */
    public abstract void talkTo(Eleanor aEleanor);

    /**
     * Specific behavior for talking to Liam
     */
    public abstract void talkTo(Liam aLiam);

    /**
     * Specific behavior for talking to Mae
     */
```
Your `Person` class includes a method for one person to talk, in a completely generically manner, to another. But then your each of your four subclasses (one per head TA) includes four additional `talkTo` methods specifying how each head TA talks to all the others (including themselves!). For example, here’s what Alex’s class (the labs head TA) looks like:

```java
package doubleDispatch;

/*
 * A class that represents Alex!
 */
public class Alex extends Person {

    /*
    * A constructor for Alex
    */
    public Alex() {
        this.name = "Alex";
        this.greeting = "Hey there! I'm Alex";
    }

    @Override
    public void talkTo(Alex aAlex) {
        super.talkTo((Person) this);
        System.out.println("Alex, I just love talking to myself");
    }

    @Override
    public void talkTo(Eleanor aEleanor) {
        super.talkTo((Person) this);
        System.out.println("Eleanor! How are homeworks? Labs are great!");
    }

    @Override
    public void talkTo(Liam aLiam) {
        super.talkTo((Person) this);
        System.out.println("Liam! How are homeworks as well? Labs are great!");
    }

    @Override
    public void talkTo(Mae aMae) {
        super.talkTo((Person) this);
        System.out.println("Mae, how are projects? Labs are great!");
    }
}
```

Now Alex can greet Eleanor in a specific manner. Likewise, she can also greet all the others in a specific manner. And they can greet her. So, are we all set then? Well, not quite. The trouble is,
someone went along and implemented the following method:

```java
/**
 * Determine a student's favorite HTA.
 * @param studentName - the name of a student
 * @return that student's favorite HTA
 */
public static Person getFavoriteHTA(String studentName) {
    if (studentName.equals("Black Panther")) {
        return new Alex();
    } else if (studentName.equals("Wonder Woman")) {
        return new Eleanor();
    } else if (studentName.equals("Wolverine")) {
        return new Liam();
    } else {
        return new Mae();
    }
}
```

The return type of this new method is, aptly, `Person`. That way, it can return any of a `Alex`, or a `Eleanor` or a `Liam`, or a `Mae`. But a difficulty arises if Alex wants to talk to someone’s favorite head TA, say `getFavoriteHTA("Wonder Woman")`. Although Wonder Woman’s favorite head TA appears to be Eleanor, the `getFavoriteHTA` method returns a `Person`. So, when Alex attempts to talk to Wonder Woman’s favorite head TA, the method that she will invoke is the generic `talkTo` method, which she uses to talk (completely generically) to a `Person`. And this is not what she wants to do!

So how do we fix this problem? That is, how can Alex talk specifically to Eleanor, while, at the same time, talking to `getFavoriteHTA("Wonder Woman")`, who she doesn’t know is Eleanor, but knows only is a `Person`? One pattern that is used to implement this functionality is called **double dispatch**. The way it works is: we flip things around. Since in the usual setup, Alex doesn’t know that she is talking to Eleanor, and so she can’t address her in the specific way that she wants to, we instead have Eleanor, who knows she is Eleanor, “be talked to by” Alex.

In code, what this amounts to is the following: in the subclass `Alex`, we override the inherited `talkTo` method—the one that talks to a `Person p`—so that `p`, which knows its own type (`Eleanor`), invokes its own method, which specifies that it should be talked to by Alex: e.g., `p.beGreetedByAlex()`. Then, in the subclass `Eleanor`, we define a method `beGreetedByAlex()` which includes the specific message Alex intends for Eleanor. Here’s what this looks like:

```java
// in Alex.java
@Overrride
public void talkTo(Person p) {
    super.talkTo(this);
    p.beGreetedByAlex();
}
```

```java
// in Eleanor.java
@Overrride
public void beGreetedByAlex() {
}
```
System.out.println("How are homeworks? Labs are great!");
}

Once again, since the head TAs do not know who each student’s favorite head TA is, they do not
know specifically who they are talking to. But the person being talked to knows who they are. So
the trick is to let the person being talked to invoke the appropriate behavior.

The double dispatch model enables a caller to defer the behavior to a callee. Since the callee knows
its own type, we need only tell the callee who the caller is, which we do by calling the corresponding
method. And voila! Our head TAs can interact appropriately, and specifically!

This code is in the /course/cs0180/src/showdown, including a well-implemented version as
well as version that makes poor use of double dispatch. If you still feel shaky on double dispatch,
feel free to play around with the example code to get a hang of it.

B Card Factory

In order to allow you to convert the String name for a class that you read in from a deck file into
an instance of that class, we have provided you with support code called the CardFactory. In
object-oriented architecture, a Factory is an object that produces objects of a specific type based
on its inputs. Our CardFactory creates an immutable map from prescribed names for the project’s
classes to the classes themselves, which you will write!

The CardFactory works by making use of reflection. Reflection is a way for imperative programs
to run code that was loaded after compile-time. This scheme allows already running programs to
include new features without needing to be shut down, recompiled, and relaunched. Rather than
compiling the code before the program is launched, reflection compiles the code at run time, which
can lead to some tricky errors, so we did our best to take care of it for you.

CardFactory exposes two methods. The first, getInstance, returns the CardFactory. (Only
one CardFactory should ever be instantiated because there is no need to duplicate the immutable
map. We call this a singleton.) The second is getDeck, which converts a list of strings into a list
of cards. All you need to do is use ShowIO to read the list of cards; then you pass that list to the
CardFactory, and voila!, out pops a list of cards.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or
any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/
courses/cs018/feedback