Lab 11: Buckets, Marbles, and Matchsticks

COSC 102 - Fall '20

In this lab you will work within a provided code base to complete the implementation of the game *Matchsticks*. You will practice working with interfaces, polymorphism, enumerated types, and will need to think critically about the data structures you have learned in this course!

1 Overview

For this lab you will be provided with a partial implementation of a tabletop game called **Matchsticks**. The game supports human players as well two different types of AIs: a simple AI and a more advanced, learning AI. Details on the game rules, the logic of these AIs, and more are outlined below:

1.1 Game Rules

The rules for the Matchsticks game are as follows:

- The game is played between **two players**.
- The players gather around a pile of matchsticks. Standard games start with **ten** matchsticks.
- A **random** player is chosen to go first.
- Each turn, the current player chooses to pick up **one**, **two**, or **three** matchsticks.
- The player to take the final matchstick **loses**.

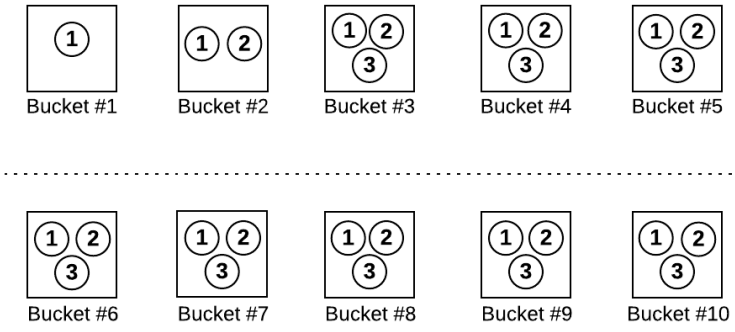
1.2 The AIs

The Matchstick program supports both human and AI (*i.e. computer controlled*) players. Furthermore, the game supports any combination of players so you will be able to challenge your AIs, or pit multiple AIs against each other!

The program supports **two different AIs**: the **basic AI** and the **learning AI**. Their logic is as follows:

- The **basic AI** simply selects a random number of matchsticks on each turn, taking no other factors into account.
- The **learning AI** is a bit more complex. Imagine we have a collection of **buckets**, one bucket for each number of matchsticks that could on the table at any point in the game. In each bucket are **marbles** labeled with a number **1**, **2**, or **3**. Each bucket starts with one marble for each of the possible choices that could be made at that matchstick total.

  If our game starts with **10 matchsticks**, the AI would have **10 buckets** with the following marbles inside:



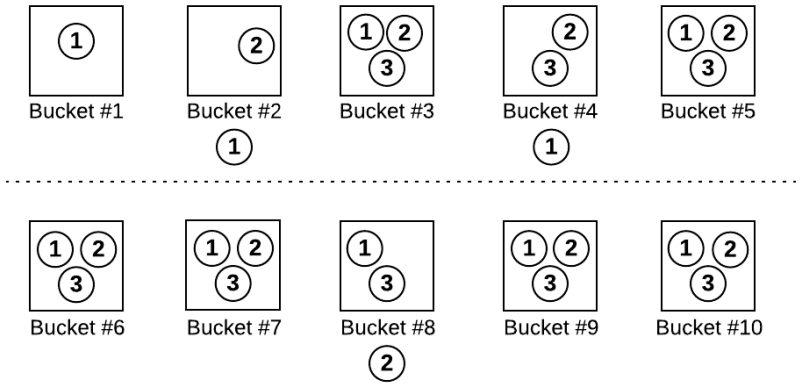
  Note that for bucket #1, there is only a single marble in the bucket (labeled with a 1), since picking two or three matchsticks at that total isn't an option. The same logic applies for bucket #2 and picking three matchsticks.

On the AI's turn, it will remove a *random* marble from the bucket corresponding to the matchsticks remaining, and set that marble beside its respective bucket. The marble drawn determines the AI's choice for that turn.

Given the above scenario, let's imagine the following:

- On the AI's first turn, there are **8 matchsticks remaining**, and it chooses to take **2 matchsticks**.
- On the AI's second turn, there are **4 matchsticks remaining**, and it chooses to take **1 matchstick**.
- On the AI's third turn, there are **2 matchsticks remaining**, and it chooses to take **1 matchstick**.

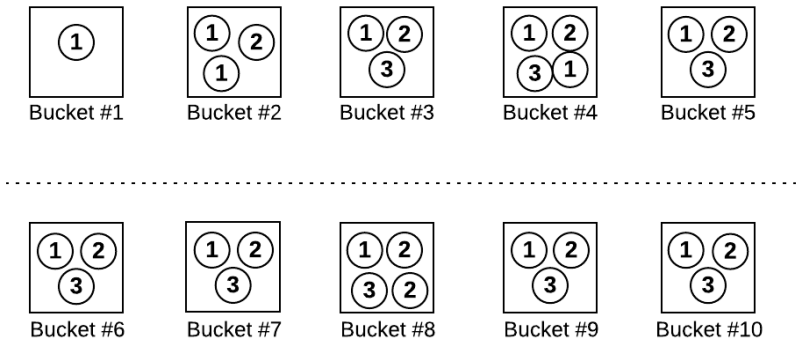
This would result in the following:



At the end of a game, if the **AI wins**, they **put back** each marble they removed into their respective bucket, **plus another marble of the same number**.

Alternatively, if the AI **loses**, they **do not put the marbles back in their respective buckets**. However, there is one exception: if removing a marble would cause its bucket to be **empty**, that marble is instead **put back**.

Since the AI in the above example won the game, at the start of the next game their buckets would look like this:



1.3 One Last Thing...

The provided code makes use of a new data type called an enumerated type (or `enum`). This is a special data type that has a finite set of predefined values. Below is an example of an `enum` representing the four cardinal directions:

```
public enum Direction{NORTH, SOUTH, EAST, WEST};
```

This creates the unique type `Direction` which can only ever equal one of the four predefined values: `NORTH`, `SOUTH`, `EAST`, `WEST`, or `null`. This newly created type can be used just like any other traditional type; this includes use with typing variables, in conditionals, and in functions as argument or return types.

To reference an `enum` value, you first must qualify it with the `enum`'s type. You can check equality against an `enum` value using == and can also use `enum` types in `switch` statements. Below is some example code:

```
public void checkDirection(Direction dir){
    if (dir == Direction.NORTH)
        System.out.println("You are heading towards Canada!");
}
```

Enumerated make code more readable while giving better control over your program's constraints. For example, in the function above, by having an argument of type `Direction`, the programmer knows that `dir` can only have one of five values: `NORTH`, `SOUTH`, `EAST`, `WEST`, or `null`.

2 Getting Started

The following sections will give you an overview of the provided classes and help you get acclimated to the starting codebase. You will be required to answer questions and draw diagrams as you familiarize yourself with this code.

2.1 Trying the Game

Before continuing on, **play a few games** of the Matchstick game as provided. You can run the game via the `main` method in the **`GameLauncher`** class. By default, the game will feature one human player and one AI player.

As you are playing, answer the following questions:

(a) How many matchsticks does the game start with?: 10 **matchsticks**

(b) The AI player is using what logic (*hint: review section 1.2*)?: basic **AI**

(c) The provided code features an optional **debug mode** which can be toggled on or off, and will display helpful information regarding each player.

Is debug mode enabled here and, if so, what information is printed for both players?:

Debug mode is enabled. It records the turns that the AI took. It's not implemented for the human player.

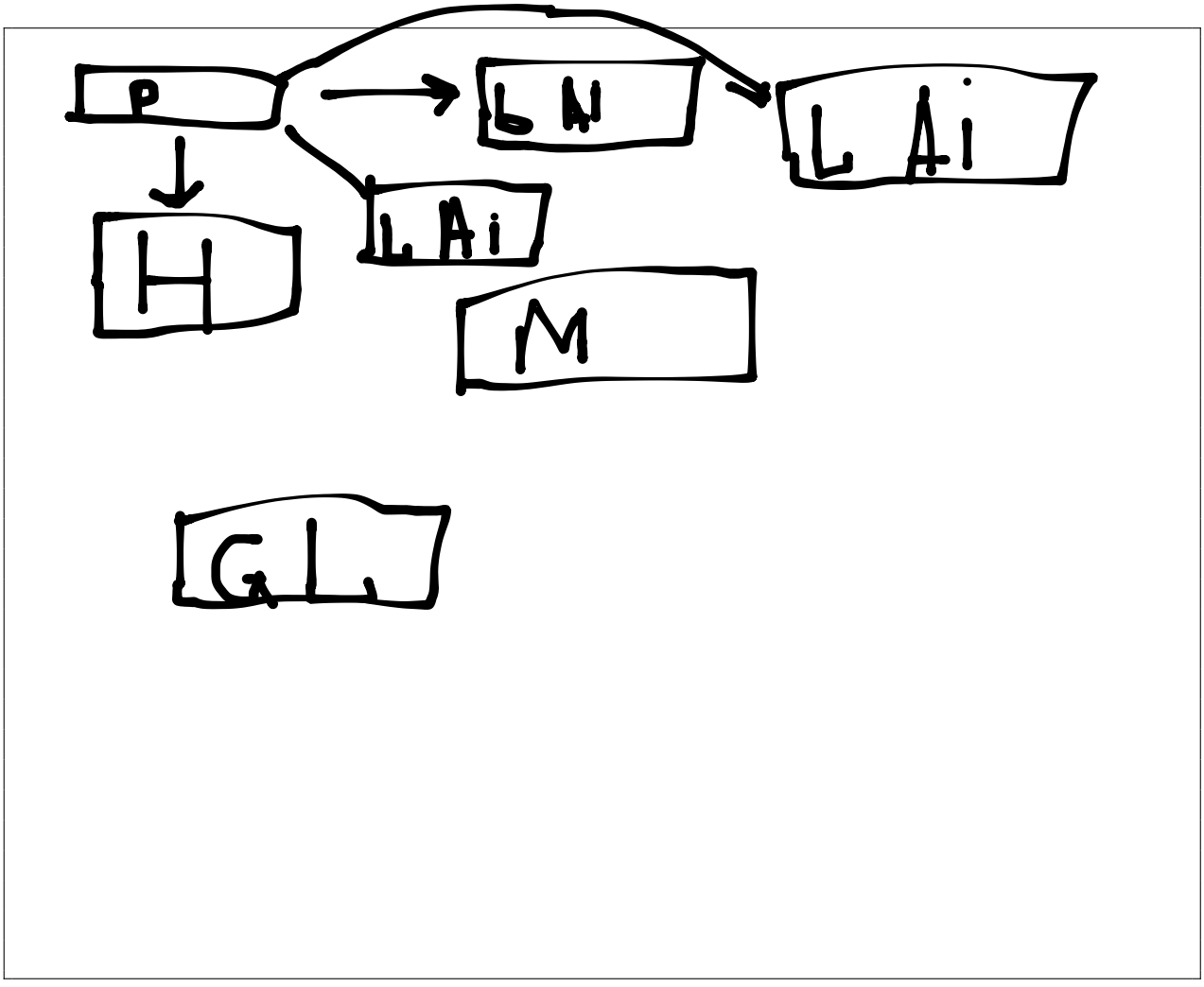
2.2 Provided Code

The Matchsticks program is comprised of **seven** classes/interfaces, one of which you will need to create. Below is a brief overview of each – classes you will need to create or modify are marked with an asterisk (*****):

- **`*GameLauncher`:** configures and launches the Matchsticks game. Contains the `main` method.
- **`Matchsticks`:** a representation of a Matchsticks game. Maintains the state of the game (including its players).
- **`Player`:** contains the required methods for objects representing some type of player of the Matchsticks game.
- **`*Human`:** a player controlled by a human.
- **`BasicAI`:** a player controlled by the computer following the basic AI logic outlined above.
- **`*LearningAI`:** a player controlled by the computer following the learning AI logic outlined above.
- **`LearningAIDemo`:** a fully functional demo version of the learning AI as described above. You are only provided with the compiled bytecode for this class.

In the box below, **draw the inheritance diagram** of these seven provided classes/interfaces.

Your diagram must include **seven** boxes, one for each item above, and must clearly indicate any which are interfaces. Draw lines to indicate which classes implement any interface(s); there is no super/sub class relationships between any of these classes. You are encouraged to look at the code of the provided files.



2.3 Tracing the Codebase

The steps below will help you get acclimated to the provided code. Complete each step in order and answer the associated questions:

(d) Start in **`GameLauncher.java`**. The `main` method here makes use of *command line arguments*. How many command line arguments are used here, and what are they used for?

Command line argument is used to determine the number of matchsticks for the game.

(e) What happens if no command line arguments are provided?

# of matchsticks will equal to 10.

(f) **`GameLauncher`** declares an enumerated type. Complete the following:

The enumerated type is declared on line: 6 and named: PlayerType

It has 4 possible values, which are:

Human, AIBasic, AILearning, AILearningDemo

This enumerated type is used in the function: In createPlayer .

and this function does the following:

It creates players. C: (what a surprise!)
It looks at the type of player and creates a player accordingly (Human/AI/etc)

(g) Next, look at **`Matchsticks.java`**. What arguments does `Matchstick`'s constructor take, and what do these arguments represent?

Type of player1, type of player2 +
number of matchsticks

(h) Trace through the **`playSingleGame`** method and then answer the following:
What governs whether the game runs in debug mode or not?

The if (DEBUG) statement

How does the `while` loop here track and alternate between the two players?

While loops runs until there are more
than 0 matchsticks, and they use mode
to alternate between players

(i) Next, look for **`getValidInput`**, also implemented in the `Matchsticks` class.
This function is **overloaded** – describe below what this means:

It means that depending on parameters
the function will behave differently.

Give a brief description of what these functions do, and how they're different:

They check for validity of input. One
method governs Strings, and the
second one is used for integers.

(j) In the **Human** class, what is the purpose of the call to **Math.min(...)**?

So that when we have 2 matchsticks on the table, the max number to choose from would be 2.

(k) Lastly, how does **BasicAI** store its previous turn data for use in the debug mode output?

It stores previous moves in ArrayList of integers called history

(l) Now, review the interface, **Player**, and answer the following:

If playing multiple games, how many times does **initPlayer** get called for each player?

2

What type does **takeTurn** return, and what does this value represent?

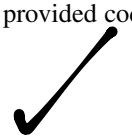
int # of matchsticks

The **gameOver** method gets called on each player at the end of each game. What does its argument represent?

It represents the boolean condition (like, if the player won or not)

3 Your Task

Now that you have a good understanding of the provided code, you are ready to finish this implementation of the matchsticks game. This will consist of two parts:



3.1 Implementing `Human`'s Debug Info

Your first task is to implement the debug mode output for **`HumanPlayer`**. `HumanPlayer`'s debug output will be similar to `BasicAI`'s – it will display all of the choices made by the player on the previous turns for the current game.

3.2 Creating the `LearningAI`

Finally, implement a learning AI player in a file named **`LearningAI.java`**, which follows the logic described in **section 1.2**. You should play games using the `LearningAIDemo` player to get a handle on how this AI operates.

You will also need to implement the debug output which works differently than the `HumanPlayer` and `BasicAI`. A `LearningAI`'s debug text displays the **buckets** as well as the **marbles in** and **on the side of** each bucket. Experiment with a `LearningAIDemo` player to understand what this output should look like.

4 Submission

Upload **only** your **`Human.java`** and **`LearningAI.java`** files to the submission link on the lab Moodle page.

This assignment is due **for all lab sections** on **Tuesday, December 8th** at **10:00PM**.