Lab 7: Hexapawn[1]
CSCI 2101 – Fall 2018

Due: Sunday, November 11, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Pair-optional

In this week's lab, you will write a program to play **Hexapawn**, a simple Chess-like game. Your program will rely on using *trees* to represent board states and will be able to automatically play the game using these trees. In completing this lab, you will also gain experience extending an existing codebase and learn about some Java features not previously discussed (abstract classes and enumeration types).

1 Hexapawn Rules

Hexapawn is a two-player game proposed by Martin Gardner in the 1960's. The game is played on a 3×3 board in which three white pawns and three black pawns are placed on opposite ends, as shown below:

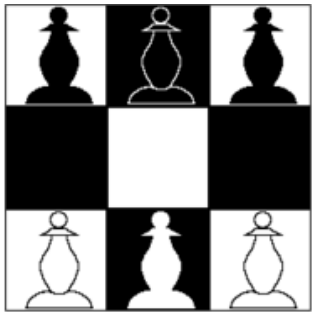

Figure 1: A starting 3×3 Hexapawn board.

Players take turns moving one of their pawns, which can be moved just like in Chess: a pawn can either be moved forward a single square (assuming the square ahead is unoccupied), or can capture an opposing pawn by moving a single square diagonally. The end-game conditions are slightly different from Chess – the game ends if any of the following events occur:

- A pawn reaches the opposite end of the board.
- A player loses all of their pawns.
- A player cannot make any further moves.

In all cases, the final player to move is the winner. Finally, note that while a standard Hexapawn board is 3×3, you can also consider a Hexapawn game on a different board size, such as 4×3 (opposing pawns are further apart) or 3×4 (four pawns on each side instead of three).

[1] Adapted from a lab provided with *Java Structures*, D. Bailey

2 Learning Hexapawn

In a 1962 article in *Scientific American*, Gardner discussed how a computer could be taught to play Hexapawn using a relatively small number of training matches. The basic idea was to keep track of the different possible states of the board and the potential for success (i.e., a win) from each state. When a particular move led directly to a loss, the computer "forgot" the move, thereby causing it to avoid that particular loss in the future. By "pruning" possible moves in this way, various intermediate game moves could indirectly lead to losses (i.e., to states that previously resulted in losses), and thus those intermediate moves would be pruned out as well. This idea is a very simple version of "reinforcement learning" which serves as the basis for today's machine learning gaming systems such as Google's Alpha-Go.

Gardner's original "computer" was constructed from matchboxes containing colored beads (this was the 1960's, remember). Each bead corresponded to a potential move, and pruning involved disposing of the last bead played. We, of course, will instead use an actual computer to run the learning process. In particular, we can store the information about each board state in the nodes of a tree, in which the root of the tree refers to the starting board state, and each child nodes refers to the state after making a particular move. Thus, the degree of each node is determined by the number of possible moves. Each leaf of the tree contains an ending configuration of the game (i.e., a winning position for one of the players and a losing position for the other).

3 Program Design

Unlike in past labs, you will not need to write all the code needed to play Hexapawn from scratch. Some classes will be provided to you, which you will use when writing others. To do so, you will need to understand the provided code and how to use it – which is typical of how most programming works in the world. You may *not* modify these classes.

3.1 Provided Classes

You will be provided with four Java classes as a starting point, which are described below. As mentioned above, you should not modify any of the provided classes.

1. `HexBoard` is a class that describes the state of a Hexapawn board. While the default board configuration is 3×3, this class can also represent an arbitrary $r \times c$ board size. Methods are provided to find all possible moves, apply moves, and check for winning board states. There is also a `main` method demonstrating how the class can be used to play a game of Hexapawn against the computer.

2. `HexMove` is a class that describes a move in a Hexapawn game, and is used by the `HexBoard` class. A `HexMove` object can be used along with an existing `HexBoard` object to produce a new `HexBoard` representing the resulting game state after making the move.

3. `Marker` is a special type of class called an `enum` that represents a marker in a game of Hexapawn (i.e., either the white or black player). Further details on `enum` types are provided in Section 5.

4. `Player` is an *abstract* class that describes a player of Hexapawn, which might be either a human or a computer player. Further details on abstract classes are provided in Section 5.

Before beginning to code, you should read through the provided class files. You can also generate and view Javadoc documentation for the provided classes by going to `Tools` and then `Project Documentation` from the main BlueJ window (once you've imported them into your new project). Make sure that you understand (at least) the public methods of the provided classes.

3.2 Classes to Write

Using the provided classes, you will have three primary tasks:

1. First, you will write a `GameTree` class representing a tree of Hexapawn positions. Note that just like our `BinaryTree` implementation, a `GameTree` object represents a single node in the tree (or, equivalently, a subtree of game positions rooted at that node). A `GameTree` node may have child nodes, but unlike a `BinaryTree`, it can have more than two (since many game positions have more than two possible moves to make).

2. Second, you will write three `Player` classes (more specifically, classes that extend the provided `Player` class) that can play games of Hexapawn. The first player should be called `HumanPlayer` and prompts the user to select moves. The second player should be called `RandomPlayer` and should play randomly. The third player should be called `ComputerPlayer` and will implement the learning procedure described in Section 2 – i.e., after each loss, the game tree will be modified to remove the losing move.

   For each of your three `Player` classes, you should have a `main` method that just plays a game between two of those players (i.e., two `HumanPlayers` against each other, or two `ComputerPlayers` against each other).

3. Third, you will write a `Hexapawn` class that allows for playing games of Hexapawn between your players. In particular, this class will allow for selecting two players (out of the three implemented above) and playing a number of games between them, then reporting how many games were won by each player. This class will just be a container for a `main` method (and any associated helper methods).

3.3 Hexapawn Interface

The interface for the final `Hexapawn` program should be modeled as shown below. In particular, you should be prompted to enter the board size (rows and columns), the type of each player (computer, random, or human – below, I've allowed them to be specified by single letters), and the number of games to play. The program should run the specified games and tally the number of wins for each player, then print out the respective win tallies, starting with player 1 (the first player to move in each game – Alice in the example below).

```
Enter number of rows: 3
Enter number of columns: 3
Enter player 1 type [crh]: r
Enter player 2 type [crh]: r
Enter number of games: 10
    [lots of game output here]
Alice won 6 games
Bob won 4 games
```

The actual textual output while your games are played doesn't need to follow any particular specification, but will probably (at a minimum) involve printing each board position as you reach it and likely the moves chosen at each step. This printing of game progress would all be happening via your `Player` classes, not within the `Hexapawn` class itself.

4 Implementation Plan

Here is a suggested plan of action for tackling the program.

- Run the main method contained in `HexBoard` a couple of times to play games against the computer. Pay particular attention to the game interface, as you will want to roughly follow it when writing your `HumanPlayer` class. Note that the computer player in the `main` method of `HexBoard` plays randomly. You can feel free to modify the board size if you want to try playing a larger game.

- Implement the `GameTree` class. This class should have a constructor that is given a `HexBoard` (the current board) and the `Marker` for the active player (i.e., the next player to move). The constructor should generate all possible boards reachable from the current board position. This process should be recursive, in that the `GameTree` constructor will be creating other `GameTree` objects representing child board positions, which will in turn create their own child positions, and so forth. In doing so, the entire tree of board positions will be built.

  Note that alternate levels of the tree represent boards that are considered by alternate players, and leaf nodes represent board positions that are wins for the last player that moved. The latter is particularly important – remember that a lack of possible moves is one condition where a win has occurred (i.e., a base case), but it's not the *only* way to reach a winning position. As a result, you shouldn't simply rely on a lack of possible moves to decide when to cut off at a leaf. The standard 3×3 board should produce a game tree with 252 nodes.

  As usual, it's a good idea to plan out your `GameTree` methods in advance, and depending on what you write now, you might have to come back later and add more methods.

- Implement the `HumanPlayer` class, which will primarily consist of writing the `play` method. Your `play` method (for any player, not just the `HumanPlayer`) will essentially operate by making a move, then handing off play to the opposing player by calling `play` on the opponent. This makes the `play` method recursive, though it might not feel like the sort of recursion you're used to. Note that in order to prevent infinite recursion, you need to separately identify when the game is over (i.e., the base case) – there are multiple ways to do this, but a good approach is to check whether the current player has *lost* before trying to move at all.

  A game, therefore, will consist of a series of recursive `play` calls between opponents. Ultimately, one of these calls will result in the game ending and will return the winning player. That player will be passed back through the recursive chain to your original calling method.

  The interface for the `HumanPlayer` should be modeled on the interface used by the `main` method in `HexBoard` (i.e., print all possible moves and then prompt to select one of them) – feel free to take snippets of code from that method and reuse them, particularly for your interface.

To test, write a `main` method in `HumanPlayer` that just plays a game between two `HumanPlayer` objects and then prints the winner at the end. This method should be quite short, since it will be relying on your `GameTree` class and your `play` method to do most of the work.

- Now, implement the `RandomPlayer` class, which will follow the same idea as the `HumanPlayer`, but will just play randomly instead of asking the user for a move selection. Implementing this should be fairly easy if you have a working `HumanPlayer` class. Write another `main` method in `RandomPlayer` to play a game between two `RandomPlayer` objects and verify by playing a couple of games (obviously, you shouldn't always get the same winner with random players).

- Now, implement the `ComputerPlayer` class. This player will operate similarly to `RandomPlayer` at first – the key difference is that the player will respond to losses by eliminating the last played node from the game tree.

  One tricky thing to watch out for is that you shouldn't eliminate *multiple* moves from the game tree when a loss occurs. To be conservative, you should only eliminate a single move at the conclusion of each game, assuming it was a loss. The reason this is tricky is that the `play` method is recursive – each subsequent game move involves another call to `play` (on alternating players), and none of these calls to `play` actually start returning until the game ends. Thus, when the game is actually lost, you're at the end of a path of recursive calls to `play` starting at the root of the game tree and going all the way down to the current leaf node. When you start returning from all of these calls, the losing player will know at each intermediate node that the game is lost, but *shouldn't* respond by pruning every move made.

  Thus, you should think carefully about how you can prune the "losing" move without pruning every other move made by that player during the game. You may find it useful to note that your recursive calls are bouncing between the same two `Player` objects (i.e., there are only two `Player` instances regardless of how deep the recursive call chain is).

  Lastly, remember to write a `main` method as with the other players that plays one game between two `ComputerPlayers`.

- Finally, implement the `Hexapawn` class (which contains the "real" `main` method). If your players are all working, this method should be fairly straightforward, and will mostly consist of providing the interface specified in Section 3.3. You will find your `Hexapawn` class very useful as you answer the questions in Section 6.

5 Implementation Advice

This lab makes use of two useful Java features that we haven't encountered in past labs: abstract classes and `enum` types. Both are described below.

5.1 Abstract Classes

The `Player` class is an example of an `abstract` class. While we've seen a few examples of `abstract` classes in the past, we've never talked very much about them. Informally, an abstract class is sort of like a hybrid between a regular class and an interface. We know that an interface defines a set of methods, but doesn't provide implementations for those methods. A regular class, on the other hand, must implement every method defined in the class.

An abstract class may have *both* methods with implementations (like a regular class) and methods without implementations (like an interface). In an abstract class, every defined but not implemented method is called an *abstract method* and must be labeled with the `abstract` keyword. Note that any class that has at least one abstract method must also be declared `abstract` at the class level (i.e., only abstract classes are allowed to have abstract methods).

We know that we can use an interface by writing a regular class that implements the interface. Similarly, we can use an abstract class by writing a regular class that inherits from the abstract class (note that since an abstract class is still a class, not an interface, we use inheritance – i.e. `extends` – instead of `implements` like we would with an interface). In order for a class extending an abstract class to be a regular (non-abstract) class, it must provide an implementation for every abstract method defined in the parent class.

The `Player` class defines a single abstract method `play`, as well as a few regular methods. As a result, in order to write an actual player, you will write a new class extending the `Player` class and provide an implementation of the `play` method in your subclass. You are free, of course, to define other methods in your subclass beyond the required `play` method if you wish. Remember that you will also need to write a subclass constructor that calls the `Player` constructor using `super` as you've done in past labs.

5.2 Enumeration Types

Quite often, we want to be able to represent a value out of some predefined set of possible values. For example, suppose we're writing a program that works with a deck of playing cards. Each card in the deck has some suit (spade, heart, club, or diamond), and we need some way to represent the card suit. One reasonable and typically-used approach is to represent each value by some arbitrary (but unique) `int`, and just define a constant for each value, such as shown below:

```
public static final int SPADE = 0; // values are arbitrary but unique
public static final int HEART = 1;
public static final int CLUB = 2;
public static final int DIAMOND = 3;
...
int myCardSuit = HEART;
```

While this approach is simple and works, it has a few downsides. The major issue is that we're representing a card suit by an `int`, but it's supposed to be just one of the four specified numbers, not any arbitrary integer. Since the variable is just an `int`, there's nothing stopping you from assigning some invalid value (e.g., 20) to `myCardSuit`, which wouldn't make sense. Similarly, the `int` type itself doesn't enforce any proper usage; e.g., if you wrote some method that accepted a card suit as a parameter, the parameter type would be `int`, but there would be no way to prevent the caller from handing the method some arbitrary value that's not `SPADE`, `HEART`, `CLUB`, or `DIAMOND`.

A more elegant way to approach this situation is to define our set of possible values using an `enum` (enumeration) type, as shown below:

```
public enum Suit {
    SPADE, HEART, CLUB, DIAMOND
}
```

Having defined this `enum` type, we can now use the new type as shown below:

```
Suit myCardSuit = Suit.HEART; // or Suit.SPADE, etc.
```

Using an `enum` has multiple benefits here. One, we now have a new type (`Suit`) that explicitly represents a card suit, which is better than just using an `int` type. Two, the code will now enforce that any `Suit` variable is one of the specified allowed values. Three, the code is cleaner, and doesn't require defining a bunch of constants with arbitrary numeric values.

While most simple `enum` types are similar to the above (a type that just has a listing of the possible values), since `enum` values are actually full-blown objects (not primitive types), you can get fancy and give them methods. The provided `Marker` class is an example of this. In addition to defining the two enum values `Marker.WHITE` and `Marker.BLACK`, the class also defines a few useful methods that can be called on `Marker` objects.

Note that the `Marker` class does have a constructor, but it's marked `private`. This reflects the fact that `enum` objects are never explicitly constructed – each possible value in the `enum` type corresponds to one instance of the class, and no other instances can ever be created in the program. In the case of the `Marker` class, there are exactly two instances (the white instance and the black instance) which are automatically constructed on startup, and no other `Marker` instances can ever be constructed after that.

6 Thought Questions

Once you have completed your program, use it to answer the following questions. Write your answers in the program `README` file.

1. Compute the total number of board positions for each of the following board sizes: (a) 3×4, (b) 3×5, and (c) 4×4. Note that as mentioned previously, there are 252 board positions for the standard 3×3 board.

2. For each of the following board sizes, determine who, if anyone, has the advantage (white or black, assuming that white moves first): (a) 3×3, (b) 3×4, and (c) 4×3. Essentially, this question is asking whether it's better to move first or second in a game of Hexapawn using the specified board size. In addition to answering the question itself, you should explain how you arrived at your answer (e.g., what your test results were).

For answering the second question, you will want to pit two computer players against each other for multiple games (Gardner called his two computers H.I.M. and H.E.R. for the first and second players, respectively). If the board advantages one of the two players, then as the computers learn to play well through successive games, you should start to see one of them gain an advantage and win more of the games.

If learning is working correctly, your results should make the advantaged player obvious as you increase the number of games. Make sure that you run enough games to see these results – e.g., 10 games is probably not enough, but 1000 games probably is. Also make sure that you're reusing the same `GameTree` object for each game – since pruning happens with each game, you don't want to lose the learning that happened in previous games by reconstructing a brand new game tree for each game.

You're welcome to experiment with larger game board sizes as well, but the number of board positions may make it too slow to collect results.

7 Evaluation

As usual, your completed program will be graded on correctness, design, and style, as well as your thought question answers included in your `README`. Make sure that your program is documented appropriately and is in compliance with the coding and style guide. Lastly, make sure that your name (and the name of your partner, if applicable) is included in both your Java files as well as your `README`.

8 Submitting Your Program

Submit your program on Blackboard in the usual way. Remember to create a zip file named with your username(s) and lab number, e.g., `sbowdoin-jbowdoin-lab7.zip`, and upload that file. Also remember to submit your group reports to me by email if working with a partner.