# Lab: Chess

In this lab, you will be creating a simple chess-like game. The lab is based on the GridWorld classes, along with several chess-related classes. ***For your convenience, you may obtain all of these files by downloading and extracting chessfiles.zip.***

## *Exercise 1: Board Already*

Create a class called `Game`, and write the following code in its `main` method:

```
Board board = new Board();
BoardDisplay display = new BoardDisplay(board);
```

Run your program. You should now see an empty chess board on the screen. Take a look at the `Board` class. You'll notice that it is a subclass of `BoundedGrid<Piece>`, and that it supports some additional behavior. You'll be adding methods to this class later in the lab.

## *Exercise 2: Return of the King*

Take a look at the `Piece` class, which deliberately resembles the `Actor` class from the case study. Each `Piece` stores a reference to the `Board` it's on, its `Location` on that `Board`, its `Color`, the name of the file used to display itself, and its value to a chess player.

Create a subclass of `Piece` called `King`. Its constructor should take everything that a `Piece` takes, except its value. A `King`'s value will always be 1000.

Now add the following code to `Game`'s `main` method (*before* you create the `BoardDisplay`!), and two kings should appear on your board.

```
Piece blackKing = new King(Color.BLACK,
                           "black_king.gif");
blackKing.putSelfInGrid(board, new Location(0, 4));
Piece whiteKing = new King(Color.WHITE,
                           "white_king.gif");
whiteKing.putSelfInGrid(board, new Location(7, 4));
```

## *Exercise 3: A King's Destiny*

Add the following method to the `Piece` class. This method should return `true` if `dest` is valid, and is either (a) empty or (b) occupied by a `Piece` of a different color.

```
public boolean isValidDestination(Location dest)
```

Now add the following abstract method to the `Piece` class. This method should return an `ArrayList<Location>` for listing the locations the `Piece` can move to. Be sure to indicate that the `Piece` class itself is now abstract.

```
public abstract ArrayList<Location> destinations();
```

Of course, you'll now need to implement `destinations` in `King`. A `King` can move to any of its 8 neighboring locations. Create an empty `ArrayList<Location>`. For each of the 8 neighboring locations, if the `Location` is a valid destination (call the method you just wrote in `Piece`), add it to the `ArrayList`. Then return the `ArrayList`.

Now go back to `Game`'s `main` method, and choose one of the kings. Loop over each of that `King`'s possible destinations. For each such `Location`, use `BoardDisplay`'s `setColor(Location, Color)` method to highlight that square on the board.

## *Exercise 4: Me Castle, You Castle*

Add the following `sweep` method to the `Piece` class.

```
public void sweep(ArrayList<Location> dests,
                  int direction)
```

This method is best explained by example. Calling `sweep(locs, Location.NORTH)` will add to `locs` all locations north of this `Piece`, until another `Piece` or the edge of the `Board` is reached. If the `Piece` encountered is of an opposing color, its `Location` is also added to `locs` (indicating that this `Piece` can capture the opposing one).

Now create a new subclass of `Piece` called `Rook`, which will have a value of 5. A `Rook` can move an arbitrary number of spaces horizontally or vertically (but not both), provided no obstacle is encountered. Use the `sweep` method to indicate which `destinations` a `Rook` can reach.

Finally, change the `main` method to add black rooks to the northern corners of the board, and white rooks to the southern corners. Light up the squares that one of your rooks can reach. Make sure these squares match your intuition for how a rook should move.

## *Exercise 5: All the Right Moves*

Take a look at the `Move` class, which describes the movement of a `Piece` to a new `Location`. (Because a `Move` can also be undone, it also keeps track of the source `Location` and any `Piece` captured in the movement.)

Add the following `allMoves` method to the `Board` class. This method should return an `ArrayList` of `Move` objects, representing all possible legal moves for pieces of the given `Color`.

```
public ArrayList<Move> allMoves(Color color)
```

Modify your `main` method to use `allMoves` in order to light up all squares on the `Board` that can be reached by pieces of one `Color`.

## *Exercise 6: Be a Player*

Define a `Player` class, which will represent one of the two players in a game of chess. A `Player` should keep track of a `Board`, the name of the `Player`, and the `Player`'s `Color`. The `Player`'s constructor should take in and store these three values. Provide methods for getting each of these values back. Also, add the following abstract method, which can be queried to get the `Player`'s next `Move` in the game.

```
public abstract Move nextMove();
```

Now create a subclass of `Player` called `RandomPlayer`, which will provide an implementation of the `nextMove` method. This method should determine all of the `RandomPlayer`'s valid moves (using the `allMoves` method you just wrote), and then return one of them at random.

Revise your `main` method to create a `RandomPlayer`, and ask it for its next `Move`. Tell the display to light up the source and destination of this `Move`. Test that a different move is lit up each time you run your code.

## *Exercise 7: Making Your Move*

Add the following `executeMove` method to the `Board` class. This method should examine `move`, and cause the designated `Piece` to move to its destination on the `Board`.

```
public void executeMove(Move move)
```

Now add the following method in `Game`.

```
private static void nextTurn(Board board,
                     BoardDisplay display, Player player)
```

The `nextTurn` method must:

1. Use the `BoardDisplay`'s `setTitle` method to show `player`'s name.
2. Ask `player` for its next move.
3. Execute that move.
4. Call the `BoardDisplay`'s `clearColors` method (removing colored borders).
5. Call the `setColor` method to highlight the source and destination of the move.
6. Use the following code to pause for half a second:

```
try {Thread.sleep(500);} catch(InterruptedException e) {}
```

Modify the `main` method to call `nextTurn`. Test that a different move is executed each time you run your code.

## *Exercise 8: A Battle of the Witless*

Add the following simple method to the `Game` class. This method should repeatedly ask the white player for its `nextTurn`, then the black player, then white, and so on.

```
public static void play(Board board, BoardDisplay display,
                        Player white, Player black)
```

Modify `main` to call the `play` method, and watch a particularly pointless game unfold.

## *Exercise 9: The Rise of Humanity*

Create a new kind of `Player` called `HumanPlayer`. A `HumanPlayer`'s constructor should take in a `BoardDisplay`, in addition to the usual `Player` parameters. Its `nextMove` method should call the `BoardDisplay`'s `selectMove` method, to prompt the user for a `Move`. Use the `Board`'s `allMoves` method to test if this `Move` is legal. If so, return it. Otherwise, call `selectMove` until a legal `Move` is chosen.

Now make one of the players in your game a `HumanPlayer`, and see if you can beat the `RandomPlayer`. (If you can't, you may need to repeat elementary school.) Now try making two `HumanPlayer`s and play a game against a classmate.

Congratulations! You've created a dull game. What will you implement next?

***Ms. Datar's most valuable tip:*** *Additional credit is not really additional credit in fact it's mandatory to* ***complete*** *the smart player and make an attempt at the smartest player to get full credit for this lab.*

## *Additional Credit (Just Kidding): The Missing Pieces*

Add pawns (value of 1), knights (3), bishops (3), and queens (9) for a complete game of chess. Use the traditional images for these pieces, or make your own. *Alternatively, have some fun, and design your own chess-like game, by making up your own types of pieces and picking images for them!*

## *Additional Credit: Get Smart!*

Make a `SmartPlayer` class. Write a method in this class called `score`, which should sum up the value of each of the `SmartPlayer`'s pieces, and subtract the value of each of the opponent's pieces. The `score` method should then return this total, which indicates how good the `Board`'s arrangement is for the `SmartPlayer`. (You may later decide to develop a better scoring system.)

In the `SmartPlayer`'s `nextMove` method, loop over each of the possible moves. For each such `Move`:

1. Execute the `Move` (using `Board`'s `executeMove` method).
2. Compute the score of the `Board` (using the `score` method you just wrote).
3. Undo the `Move` (using `Board`'s `undoMove` method).

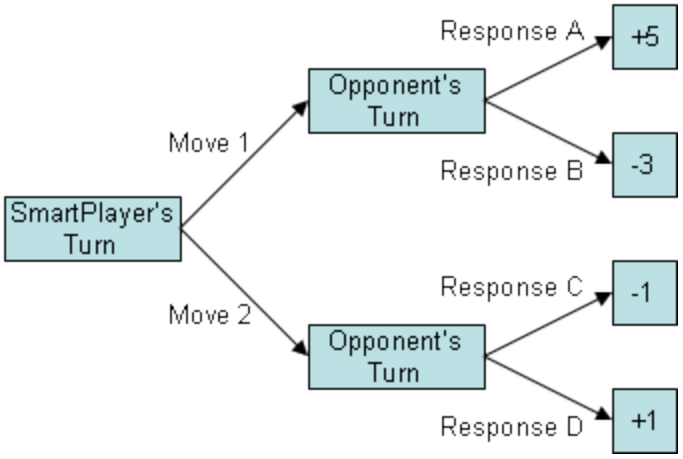
Return the `Move` that results in the highest score.

Now play a game against a `SmartPlayer`, and you'll discover that it's still pretty dim. It will capture a piece when it can, but will otherwise play as poorly as the `RandomPlayer`, leaving itself open to easy captures. If you choose to work on the next exercise, you'll find that true smarts are just around the corner!

## *Additional Credit: Get Smarter!*

(First, complete the previous "Get Smart!" exercise.)

Your `SmartPlayer` can now look one move ahead, and pick the one that's best for itself. Let's look at the simplified game tree below, in which the `SmartPlayer` must choose between 2 moves, each of which will cause the `SmartPlayer`'s opponent to choose between two response moves. For each of the four possible resulting move sequences, the game tree shows a score indicating how good this sequence is for the `SmartPlayer`. For example, if the `SmartPlayer` chooses *Move 2* and the opponent responds with *Response D*, the value of this board to the `SmartPlayer` will be +1.




Now, it turns out the best strategy is to assume your opponent is just as smart as you are, and will therefore always choose the move that is best for itself, and therefore *worst* for you. With this idea in mind, which of the two moves above should `SmartPlayer` choose, and how should `SmartPlayer`'s opponent respond?

Thus, to work out `SmartPlayer`'s best move, looking two moves ahead, it is necessary to consider all moves, and then work out the value of the opponent's meanest response. The `SmartPlayer` should then select the move that leads to the highest valued meanest response.

Modify `SmartPlayer`'s `nextMove` method to call the following helper, instead of immediately calling `score`. To implement the `valueOfMeanestResponse` method, test all moves your opponent might make, and determine the score of each such move. Return the minimum such value.

```
private int valueOfMeanestResponse()
```

You should now find that your `SmartPlayer` plays much more defensively and is not as easily beaten.

## *Additional Credit: Get Smartest!*

(First, complete the previous two exercises.)

Your `SmartPlayer` now looks ahead 2 moves, but it's just a simple matter to make it look an arbitrary number of moves ahead! Change `valueOfMeanestResponse` to take in an `int` parameter, indicating how many moves to look ahead. If asked to look ahead 0 moves, simply return the board's score. Otherwise, continue to find the value of the meanest response as before, but instead of calling `score` to rate each move, call a new `valueOfBestMove` method, which you'll ask to look ahead one less move. If asked to look ahead 0 moves, the `valueOfBestMove` method will again return the board's score. Otherwise, this new method will test all moves the `SmartPlayer` might make, and return the maximum such value. And, `valueOfBestMove` will score each such move using `valueOfMeanestResponse` again!

This is the classic Minimax algorithm, explored in the 1920s by John von Neumann, who was one of the fathers of both economic game theory and computer architecture. The Minimax algorithm is a central idea in game theory and artificial intelligence, and is the basis of all algorithms for playing chess, checkers, connect four, etc. (Notice that your `SmartPlayer` code makes no reference to kings, rooks, the rules of chess, etc!)

Your `SmartPlayer` will now be smarter than your teacher. Give it an A+, and send it to Stanford!

## *Additional Credit: Our Feature Presentation*

Add support for various chess features, or for features of your own design:

- Detecting "check" and "check mate".
- Promoting pawns that reach the other side of the board.
- Castling.
- A networked player.
- Locations on the board where pieces can teleport.
- Anything you like!