Chess (50 points)

The purpose of this project is to make (at least part of) a playable chess program. There will be a minimum level of completion, spelled out here, but how far you want to take this is up to you. Basically, what we want the program to do is:

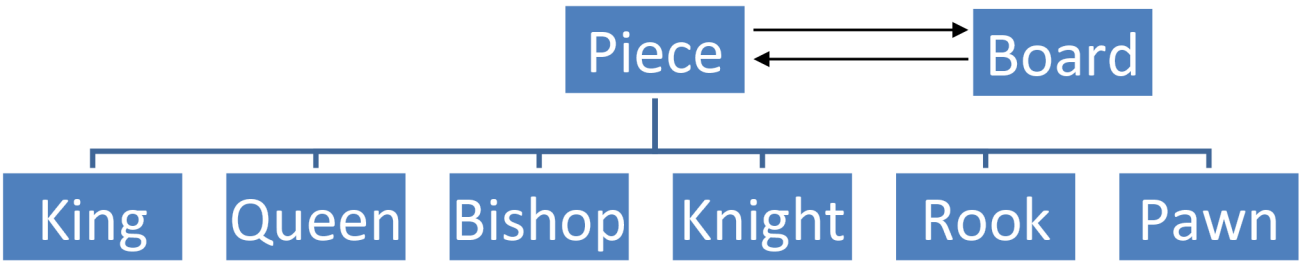
- Set up a board
- Populate it with pieces
- Move pieces from one space to another
- Determine if a move by one player is legal

To do this, you're going to write the following classes (plus one other class, listed later):

- `Board`, which will act as the game board. It will primarily act as a container for `Piece` objects, but will also handle things like printing the game board.
- `Piece`, which is a general class for the game pieces. It will have properties and methods that are common to all game pieces. The `Board` will contain a 2-D array of type `Piece`.
- Six different classes, corresponding to types of pieces:
  - `King`
  - `Queen`
  - `Bishop`
  - `Knight`
  - `Rook`
  - `Pawn`

The structure of the classes will be as follows. The arrows indicate that each class has the other class as a field/property. This is known as a "has-a" relationship. In this case, the board maintains a 2-D array of `Piece` references, which indicates that the board "has" a set of pieces. In addition, each `Piece` keeps a reference to the `Board` in which it resides. This is because it is the responsibility of the `Piece` to determine if a particular move is legal.

The hierarchy/tree diagram below `Piece` indicates that the other 6 types are subclasses of `Piece`. These classes have an "is-a" relationship with class `Piece`. Note that, for example, a `Queen` is always a `Piece`, but a `Piece` is not always a `Queen`. This explains why references of type `Piece` may legally refer to an object of any subclass type, but not the other way around. The fact that all the piece types are subclasses of `Piece` makes it easy to keep track of the board, since the array only needs to contain `Piece` references.

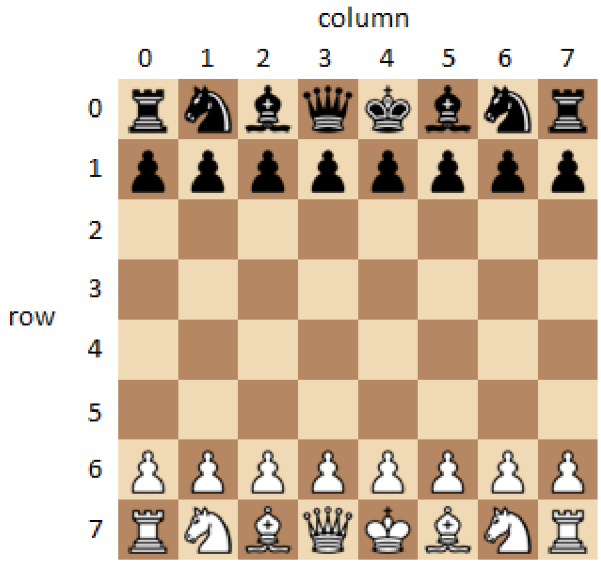


In addition, there will be one other class, called `Location`. This class will simply store a location on the board (row and column). This will be useful for methods that might need to return a location on the board, and so need to return two numbers.

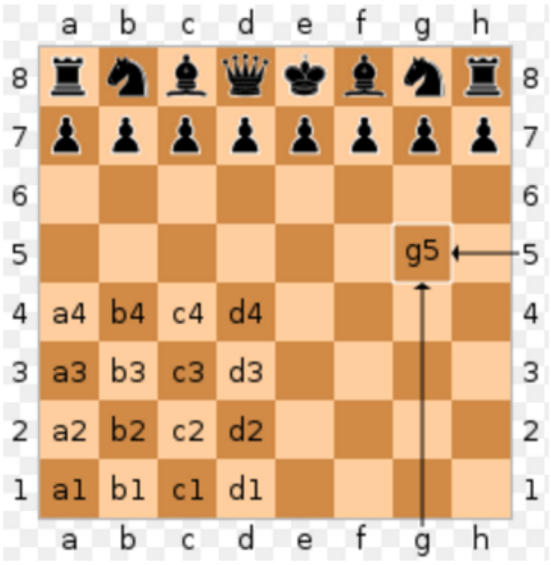
Conventions

First of all, we're going to make the rather absurd assumption that you know how to play chess. For those of you that don't, here's a link to a rather good web page on the topic: https://www.chess.com/learn-how-to-play-chess. The important part of the rules is how each piece moves.

Here's how we will set up the board in the `Board` class:



Contrast this with how conventional chess notation works:



In these notations, the black queen would be at location d8 on the board, or at `array[0][3]` in the field of the `Board` class.

Let's look at the specifications of the classes. These are the methods that have to be included. You are free, of course, to write other methods if you wish, to help with your code.

`public class Location`

Properties/fields: `row`, `column` (both `int`s)

Signature	Return type	Action
`Location(int, int)`	None	Constructor. Initializes fields.
`Location(String)`	None	Constructor. Initializes fields using algebraic notation according to the second board shown above. Parameter must be 2 characters long. The first must be a letter in the range from a to h, inclusive, case-insensitive. The second must be a number in the range 1 to 8. Any deviation from this will throw an `IllegalArgumentException`. Example: a call `new Location("d8")` would initialize the new object's fields such that `row` equals 0 and `column` equals 3.
`getRow()`	`int`	Accessor for the row field.
`getCol()`	`int`	Accessor for the column field.
`setRow(int)`	`void`	Mutator for the row field.
`setCol(int)`	`void`	Mutator for the column field.
`toString()`	`String`	Returns a string representation of the location.
`equals(Location)`	`boolean`	Returns true if the parameter is the same board `Location`

`public class Board`

Properties/fields: `array` (a 2-dimensional array of `Pieces`)

Signature	Return type	Action
`Board(boolean)`	None	Constructor. The boolean indicates whether or not the board should be reset. If the parameter is `true`, in addition to instantiating the array, the array should also be reset to standard opening position (see first diagram, previous page). This can be accomplished by calling the `reset` method (see below). If the parameter is `false`, instantiate the array and leave it empty.
`getPiece(Location)`	`Piece`	Returns a reference to the `Piece` at the specified location on the board, leaving the array unchanged. If location is empty, returns `null`.
`placePiece(Piece, Location)`	`Piece`	Puts the passed `Piece` at the specified location. If the location is occupied, the method returns the piece previously at the specified location. If the specified location was previously empty, returns `null`.
`toString()`	`String`	Returns a `String` representation of the board. This will contain 8 lines of text, so include `\n` at appropriate points in the return value. The format can be in one of the two forms shown below. You can decide which version by the way you code the `toString` methods of the various `Piece` subclasses.

Text version:

```
RNBQKBNR
PPPPPPPP
........
........
........
........
pppppppp
rnbqkbnr
```

note: the knight is abbreviated N, not K.
Color is indicated by uppercase/lowercase.

Unicode version (not sure if this works in Dr Java):

♖♘♗♕♔♗♘♖
♙♙♙♙♙♙♙♙

♟♟♟♟♟♟♟♟
♜♞♝♛♚♝♞♜

note: the pieces are generated by using Unicode characters. You can find them all at https://en.wikipedia.org/wiki/Chess_symbols_in_Unicode
To use Unicode in a `String` or `char`, type \u#### where #### is the hexadecimal Unicode for that character. For example, a black chess rook would be written \u265C.

[note: at the time of writing this, there is a problem with using the format at right. You need to use a Unicode font, such as Lucida Unicode, to print the chess pieces. However, I have not yet figured out how to handle the spaces. I need to find a space or unassuming character that is the same width as the piece characters. Help?]

`Board` (continued)

`isEmpty(Location)`	`boolean`	Returns `true` if the specified location is empty, and `false` otherwise.
`reset()`	`void`	Resets the board to the standard starting position for a chess game. This method does not instantiate the array, but should clear out all the spaces and instantiate new pieces to set up the board. (Why new pieces? Read the descriptions for the `Rook` and `King` classes.)
`locationOf(Piece)`	`Location`	Searches the board for the `Piece` passed as an argument. Returns the `Location` of the `Piece` on the board, and `null` if not found. Note that this method should use == to compare `Pieces`. For example, if there are multiple white pawns on the board, the method should return the location of the specific `pawn` which is passed to the method.
`movePiece(Location source, Location destination)`	`Piece`	Moves a piece from the first location to the second. If the destination space is empty, the method returns `null`. If the destination space is occupied, the piece currently occupying the destination space is removed and returned by the method. If the source space is empty, the method throws an `IllegalArgumentException`.
`removePiece(Location)`	`Piece`	Removes the piece from the specified location and returns the piece that is removed. If the location is empty, returns `null`.
`isValid(Location)`	`boolean`	Determines if a `Location` is a valid board location. Although the second `Location` constructor cannot accept an invalid location, the first one can. So, if the `Location` in question is not on the board, the method must return `false`.

`public abstract class Piece`

Properties/fields: `white` (type `boolean`), `myBoard` (type `Board`)

Signature	Return type	Action
`Piece(boolean, Board)`	None	Constructor. Initializes fields.
`Piece(boolean, Board, Location)`	None	Constructor. Initializes fields and places the piece on the board in the specified location.
`isWhite()`	`boolean`	Accessor method for the `white` field.
`getMyBoard()`	`Board`	Accessor method for the `myBoard` field.
`abstract move(Location)`	`boolean`	Attempts to move this `Piece` to the specified location. If such a move is legal, the piece is moved and the method returns `true`. If the move is illegal, the piece is not moved and the method returns `false`. This method is abstract and is implemented in the subclasses, with each method deciding on the move's legality based on the type of the piece.
`myLocation()`	`Location`	Returns the location of this piece within `myBoard`, by making a call to `locationOf`. If this piece is not on the board, the method will return `null`.

`public class Pawn`
`public class Knight`
`public class Bishop`
`public class Queen`

No additional fields/properties. Note the table below is for `Pawn`. Other classes are similar.

`Pawn(boolean, Board)`	None	Constructor. Initializes fields.
`Pawn(boolean, Board, Location)`	None	Constructor. Initializes fields and places the pawn on the board in the specified location.
`move(Location)`	`boolean`	See the `Piece` specification for this method.
`toString()`	`String`	Proper representation of piece based upon color

For the knight, bishop, and queen, you need to check if the destination space is reachable by the piece. You need to check if the move is blocked by any other pieces (not applicable to the knight). You need to check if the destination space is occupied—if it is, the move is legal only if the piece is of the opposite color.

The pawn is a bit more complicated. Although there is a very small selection of spaces that the pawn can legally move into, the moves cover a number of different cases. First of all, remember that white pawns move up the board, black pawns move down the board. In addition, they move diagonally when capturing. You might just want to check the three possible moves a pawn can make:

1. It can move one space away from its owner if the space is unoccupied.
2. It can move two spaces away from its owner if it's on its starting space and if the two spaces in front of the pawn are unoccupied.
3. It can move diagonally one space away from its owner, in either direction, if the space is occupied by a piece of the opposing color.

Note that when a piece moves onto a piece of the opposite color, the opposing piece is removed from the board.

`public class Rook`
`public class King`

Additional fields/properties: `hasMoved` (type `boolean`). This property tracks whether or not the piece has moved at any point in the game. Note the table below is for the `Rook` class. The king's class is similar.

`Rook(boolean, Board)`	None	Constructor. Initializes fields. Sets `hasMoved` to `false`.
`Rook(boolean, Board, Location)`	None	Constructor. Initializes fields and places the pawn on the board in the specified location. Sets `hasMoved` to `false`.
`move(Location)`	`boolean`	See the `Piece` specification for this method. If the move is legal, sets `hasMoved` to `true`.

In addition to their regular moves, this class must also handle castling. The way that a castling move is indicated is by attempting to move the king two spaces to the left or right. In order for the castling to be legal, the following must be true:

- The king must never have moved before this turn.
- The rook with which the king is attempting to castle must also never have moved before this turn.
- All the spaces between the king and the rook must be empty.

When castling, the king moves two spaces to the side and the rook moves into the space that the king passed over.

Running your program

Your driver class must do the following, at a minimum:

It must keep track of whose turn it is to play. White goes first.

On each turn, it needs to print the board to the screen. For initial testing, do this in text format (since the `Board` class has a `toString` method, this only requires printing the `Board`.) Later, if you have time, you can construct something in Swing with a GUI.

For each turn, request a source square and a destination square. It's easiest to do this in algebraic notation (as in d2 and d4) since the `Location` class accepts this notation in one of its constructor methods. Since said method throws an exception if the location is invalid, you can catch the exception and print a message if the input is invalid.

Attempt to make the move. If it is valid, go to the next turn (back to the top of this list). If it is illegal, print a message and ask for another move.

Don't worry about figuring out when the game is over (see optional extras on the next page).

Optional Extras

In the interest of time, there are a number of parts of the game of chess which we did not cover. To make this a fully-functional chess program, here are the things you would need to add:

- Check. It is illegal for any move to be made which would result in the king being in check. One way to implement this would be to write a method (in the `King` class) which looks to see if the king is in check. After a player makes an otherwise-legal move, if that player's `King` is in check, undo the move and have the `move` method return `false`.
- Castling. The rule above indicates that it is illegal to castle if the resulting move would result in the king ending up in check. However, it is actually illegal for the king to castle into, out of, or through check. So, if the king is currently in check, it may not castle. In addition, if the "middle" space (the one the king passes over in castling) is threatened by an opposing piece, the move is still illegal.
- Pawn promotion. When a pawn moves into the last row of the board, it may be changed into a knight, bishop, rook, or queen—player's choice. It is legal to promote the pawn to any type of piece, even if none of that type have previously been captured. For example, a player may have two queens or three knights on the board.
- *En passant*. Look this one up yourself online. It's a very special move by which a pawn may capture another pawn. This would require adding a way of keeping track of whether or not the captured pawn moved on the immediately-previous turn.
- Checkmate. If the king is in check and any move by the player results in the king being in check, the player loses and the game is over.
- Stalemate. If the king is not in check, but no legal move may be made (either because nothing can move or because any move would result in the king ending up in check), the game is over and is declared a draw.

For the purposes of the last two, it might be useful to add a method to the piece classes which returns an array of `Location`s to which that piece may move.