Due Oct 5 by 11:59pm **Points** 100 **Available** after Sep 25 at 9pm

Module 3: More Complex Forms of Data

Home

Modules

(https

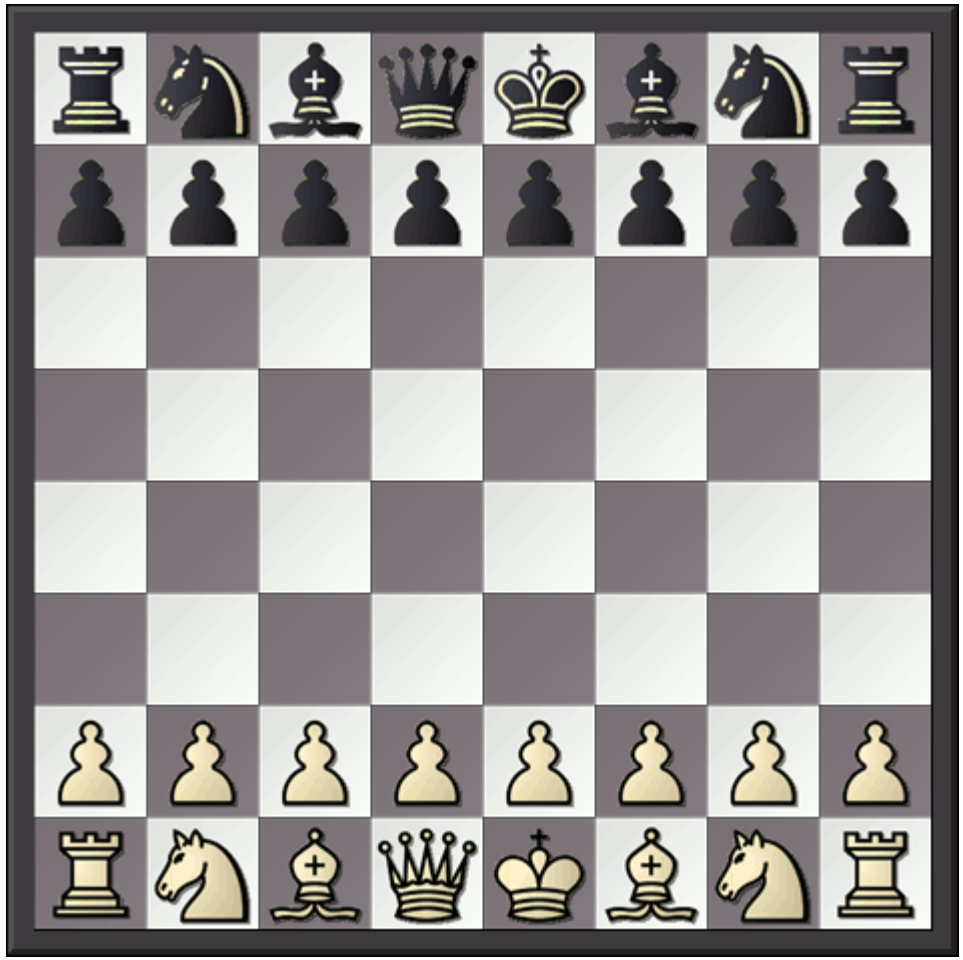
Homework 3: Chess pieces

Chess pieces

This homework will give you practice with the concepts of types, sub-types and polymorphism. As you'll learn in lecture, sub-typing is not necessarily sub-classing (for code reuse), and this homework assignment will help you begin to distinguish between those concepts. You will also learn the <<*implements*>> relationship as defined by both the UML and programming languages like Java.

Introduction

A game of chess has several kinds of pieces: pawns, knights, bishops, rooks, queens and kings. These pieces are arranged on a chess board as shown in the figure below.



(Image taken from http://www.chess-game-strategies.com/images/kqa_chessboard_large-picture_2d.gif)

A cell on the board is specified by a (row, column) tuple: rows increasing from bottom to top and columns increasing from left to right. Traditionally the black pieces are arranged in the top two rows as shown.

Each chess piece can move in a specific way. In addition to moving, each chess piece can also kill/capture a chess piece of the opposite color if it moves to its place. The rules for each chess piece are as follows:

- **Bishop**: A bishop can only move diagonally, and kill any opponent's piece if it can move to its place.
- **Knight**: A knight can move only in an L pattern: two cells horizontally and one vertically or vice versa. It can kill any opponent's piece if it can move to its place.
- **Queen**: A queen can move horizontally, vertically and diagonally. It can kill any opponent's piece if it can move to its place.
- **King**: Similar to a queen, a king can move horizontally, vertically and diagonally. However, kings are limited in that **they can only move one space at a time**, in any of those directions. It can kill any opponent's piece if it can move to its place.
- **Rook**: A rook can move horizontally or vertically. It can kill any opponent's piece if it can move to its place.
- **Pawn**: A pawn is interesting: it can move only "ahead," not backwards towards where its color started. It can move only one place forward in its own column, except for first moves from its "game start row" (row 1 for White pawns; row 6 for Black pawns) - in which case it can (optionally) move two places forward. In other words, a pawn can opt to move one square or two squares forward from its game start. Additionally, to kill it must move one place forward diagonally (it cannot kill by moving straight).

What to do

You must design and implement classes that represent the above chess pieces. Create a `chess` package and place all of your source assets in this package. **Do NOT include your JUnit test assets in the chess package**. You may place your test assets in a different package, or in the default test sources root. Whatever approach you take, organize your solution such that your test code imports your chess package.

Your classes should be named **exactly** as their names above, and all of them should implement the `ChessPiece` **interface**. Unlike previous assignments, we're not explicitly giving **all** of the the method signatures to you (we're giving you some, but not all

signatures below); rather, you must read the spec carefully and extract the signatures from the English description given. As a first step, develop the `ChessPiece` interface based on the following contract *(NB: ensure your ChessPiece interface is spelled the same way as you see here)*.

ChessPiece contract**:**

Each chess piece should be able to:

(a) return (answer) its current position on the chess board, as methods `getRow()` and `getColumn()`

(b) return (answer) its color as an `enum Color`

(c) determine and answer if it can move to a given cell, as a method `canMove(int row, int col)`,

(d) determine and answer if it can kill a provided piece starting from where it currently is, as a method `canKill(ChessPiece piece)`. In addition, each chess piece should have a constructor that takes in an initial position as a row and column, and a color as an enum `Color` with values `BLACK` and `WHITE`.
Note: All rows and columns begin at "index" 0. Rows decrease in number from top to bottom in the above chessboard. Therefore the "top" row shown on the graphic above (where the Black non-pawn pieces are positioned) is row 7 and the right most column is column 7. If there is an attempt to create a chess piece outside of these bounds, raise an `IllegalArgumentException` upon construction.

Note 2: Take note of the "special case" for the `Pawn` with respect to moving and capturing (kill) other pieces. One additional special case for Pawns for this assignment: No pawns may be created in the "royal" row for their color. In other words, White pawns may not be created in row 0 and Black pawns may not be created in row 7. Raise an `IllegalArgumentException` if either of those constraints are broken.

Beyond the classes, enum, interface and methods specified here, you are free to design any other interfaces and classes as you see fit. However, they must have good justification, be designed appropriately, and be used in the appropriate place.

You must write tests for each chess piece! Test your operations thoroughly.

Clarifying Notes:
This is a "move the pieces around the chessboard" assignment, NOT a "let's build a complete chess game" assignment.

*We are **not** considering the "board setup".*
*We are **not** considering what happens if pieces obstruct a "move path".*
Simply answer the question: it a "is this a valid move for this piece, assuming there are no other obstructing pieces on the board"? In other words, given what you know about the variation in behavior for moves, just determine if the move to a new location is valid. Or, determine if the capture/kill is valid assuming one other piece (friend or foe) is in the "target" location.

How to submit

1. Upload your .zip file to **Gradescope**. Or, you may point Gradescope to your GitHub repository.
   Reminder: If you use GitHub, your repositories for this course must remain private for the duration of this class to adhere to our academic honesty policy!
2. Wait for a few minutes for feedback to appear, and take corrective action if needed.

Other Tips

1. Our lectures covered supertype-subtype relationships and also touched on superclass-subclass code reuse. As you design this solution, take note of where those concepts may be utilized and in particular, if you can make use of Abstract classes. Remember to stay D.R.Y.