Programming Project 3

Due Friday, October 30 at 11:59pm (EDT)

IMPORTANT: *Read the* ***Do's and Dont's*** *in the* ***Course Honor Policy*** *found on blackboard.*

I. Overview

The purpose of this project is to give you practice designing a class/type hierarchy and using an API. It is *important* that you spend time designing your class hierarchy *before* you start coding. If you properly organize your classes and/or interfaces, you can achieve the desired program behavior below with significantly less code than with a poorly organized hierarchy.

II. Code Readability (20% of your project grade)

To receive the full readability marks, your code must follow the following guideline:

- All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
- All variable and method names must start with a lowercase letter. All class and interface names must start with an uppercase letter.
- The class body should be organized so that all the fields are at the top of the file, the constructors are next, the non-static methods next, and the static methods at the bottom.
- There should not be two statements on the same line.
- All code must be properly indented (see page 644 of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
- You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
- There must be an empty line between each method.
- There must be a space separating each operator from its operands as well as a space after each comma.
- There must be a comment at the top of the file that includes both your name and a description of what the class represents.
- There must be a comment directly above each method (including constructors) and states *what* task the method is doing, not how it is doing it. The comment should include tags for any parameters, return values and exceptions, and the tags should include appropriate comments that indicate the purpose of the inputs, the value returned, and the meaning of the exceptions.
- There must be a comment directly above each field that, in one line, states what the field is storing.
- There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column.
- There must be a comment above each loop that indicates the purpose of the loop. Ideally, the comment would consist of any preconditions (if they exist) and the subgoal for the loop iteration.
- Any code that is complicated should have a short comment either above it or aligned to the right that explains the logic of the code.

III. Program Testing (20% of your project grade)

New for this assignment: you should provide a JUnit test class for the project.

You are to write a test report that indicates the *kinds* of tests needed to thoroughly test your project. The tests should demonstrate that all of your methods behave correctly. An conditional statements will need tests that go through each branch of the execution. Any loops will need tests that cover the "test 0, test 1, test many" and "test first, test middle, test last" guidelines. Your testing report should *not* list the actual tests and results.

You are to have a JUnit test class or classes that implement each of the needed tests. Comments and method names in your JUnit class should connect to your testing report. For example, if your testing report states "*method xxx must be tested with string inputs of different lengths*", then the reader should be able to go to the JUnit class and easily identify the tests that test that method on inputs of length 0, 1, and more than 1.

The testing report must be separate from the JUnit class. In most companies, the testing document will be written in a style that allows both programmers and non-programmers to read it and recognize whether all the needed test cases were included.

Routines you do not have to JUnit test: A routine that displays to the screen or gets input from the user cannot be easily tested with JUnit. For these methods, your testing report should still state how you will test them, and then you will test them yourself by running your code (similar to what you did for previous projects).

Testing inherited routines: You are not *required* to have tests for methods that a class inherits but does not override. However, many companies will require you to write tests for them. The reason is that later updates may choose to override the methods and you want the tests already there for when that happens. A very good practice is to write your tests *before* you design your program based on what the desired final results are, and then the tests will verify that the classes perform correctly regardless of how you decide to create your hierarchy.

IV. Java Programming (60% of your grade)

The purpose of the project is to create an (almost) working chess game. However, we are going to take advantage of Java and design the game a little differently. We are going to make each piece encode its own rules for what is and is not a legal move. This way we can (and will in a future project) be able to do things like add new pieces to the game without redesigning the game display, or change the game board without having to redesign the pieces.

Backgound Information

This part of the instructions is the API for the clases you are provided. You can skip this section and jump down to the "What You Must Do" section to see what your design and programming task is. Then you can refer back to this section when you are ready to code. You can also get a web page with the API for these classes but loading the classes into your IDE and clicking the "JavaDoc" button/

You are provided with four types: `ChessBoard` is a generic chess board written using Java Swing, `ChessGame` is an interface for encoding general rules for how the game is played (such as the requirement that play alternates between two players). Both these are very generic so we could use them to encode almost any chess-like game, but for this project we will stick to the standard rules of Indo/European chess. `ChessBoardDisplay` is an interface that has routines describing how we want the chess board to look like, and `EuropeanChessDisplay` is an implementation of the `ChessBoardDisplay` interface.

ChessBoard API:

The chessboard draws a chessboard in a Java swing JFrame, allows pieces to be placed on the board, and allows players to click on a piece and click on the spot where the piece should move. **You do not have to modify the ChessBoard class.**

The way the chessboard works is as follows. ChessGame that determines the type of game that is being played. When the user selects a piece, the chess board calls the `isLegalPieceToPlay` method of `ChessGame` to see if the player is allowed to select that piece. If they are, that piece is highlighted and the user can then select a square, the chess board then call's the `makeMove` method of the `ChessGame` with the piece the player wants to move and the space the player wants to move the piece to. It is up to the `ChessGame` class to actually make the move.

- **Constructors**
  - **`ChessBoard(int numRows, int numColumns, ChessBoardDisplay display, ChessGame gameRules)`**: Creates a chessboard with the given number of rows and columns. The `ChessBoardDisplay` indicates how the squares of the chessboard should be drawn, and the `ChessGame` is used to determine if a selected move is legal and then to make that move.
- **Methods**
  - **`ChessGame getGameRules`** gets the object that is controlling how the game is being played.
  - **`void setGameRules(ChessGame newRules)`** provides an object that controls how the game is played on the board.
  - **`int numRows()`** returns the number of rows of the board.
  - **`int numColumns()`** returns the number of columns of the board.
  - **`void addPiece(ChessPiece piece, int row, int column):`** Adds a `ChessPiece` to the board at a specified row and column
  - **`ChessPiece getPiece(int row, int column):`** returns the `ChessPiece` that is at the specified row and column of the board. Returns `null` if that board location is empty.
  - **`boolean hasPiece(int row, int col):`** returns `true` if there is a piece at the specified row and column.
  - **`ChessPiece removePiece(int row, int col):`** Removes a `ChessPiece` from the board. Returns the removed piece.
  - **`boolean squareThreatened(int row, int col, ChessPiece piece):`** returns `true` if the chess board square at the specified row and column is threatened by a piece of the *opposing* side as `piece`. A square is threatened if a piece can move there in one legal move.

ChessBoardDisplay API:

The ChessBoardDisplay controls how the different squares on the chessboard are drawn. **You do not have to modify the ChessBoardDisplay interface.**

- **Methods**
  - **`int getSquareSize()`** returns the number of pixels per side for a square on the chessboard.
  - **`void displayEmptySquare(JButton button, int row, int column)`** sets how an empty square should be displayed. The input is the button being used for the square and the location of that square on the board.
  - **`void displayFilledSquare(JButton button, int row, int column, ChessPiece piece)`** sets how a square containing a piece should be displayed. The input is the button being used for the square, the location of that square on the board, and the piece on that square.
  - **`void highlightSquare(JButton button, int row, int column, ChessPiece piece)`** sets how a square should be displayed if the board is "highlighting" the square. The input is the button being used for the square, the location of that square on the board, and the piece, if any, on that square.

EuropeanChessDisplay API:

This class implements the ChessBoardDisplay interface to create a board that is a red and black alternating grid, and it uses yellow and green as the colors of the chess pieces for a north/south game and white and gray for the color of the pieces in an east/west game. **You are welcome to change this class if you want to change how the game looks, but you are not required to.**

ChessRules API:

The ChessRules control the basic way that the game is played. This type *does not* encode how pieces should move. Instead, it will control basic rules such as which player moves next. **You should not modify this interface.**

- **Methods**
  - **`boolean legalPieceToPlay(ChessPiece piece, int row, int column)`** returns whether the player is permitted to move this piece. The inputs are the chess piece the player selected and the location on the board of the piece.
  - **`boolean makeMove(ChessPiece piece, int toRow, int toColumn)`** this is called if a wishes to move a piece to a new location. The method should determine if the move is legal, and if it is, update the game and return `true`. If the move is not legal, it should return `false`.
  - **`boolean canChangeSelectin(ChessPiece piece, int row, int column)`** returns whether the user is allowed to change the piece they selected to move. The default is `true`. Returning false will require the player to choose a valid square to move the piece. *Warning*: if you return false, then the `legalPieceToPlace` method must not return `true` on a piece with no legal moves; otherwise, this will freeze the game.
- **Nested Types**
  - The ChessRules interface contains an *enum* called `Side` that is used to indicate the different *sides* (or *players*) in the game. The players are named by their starting locations on the board: `NORTH, SOUTH, EAST, WEST`. For a standard 2-person game, you will either make the players `NORTH` and `SOUTH` or `EAST` and `WEST`. You can access the enum values with `Side.NORTH` inside the `ChessRules` hierarchy or `ChessRules.Side.NORTH` outside of it.

An overview of enum

The project will be using enum types. An *enum* is a shortcut for a class with a private constructor. The code

```
enum WeekDay {
  Monday, Tuesday, Wednesday, Thursday, Friday;
  you may add additional methods here
}
```

is identical to

```
public static class WeekDay {
  public static final WeekDay Monday = new WeekDay();
  public static final WeekDay Tuesday = new WeekDay();
  public static final WeekDay Wednesday = new WeekDay();
  public static final WeekDay Thursday = new WeekDay();
  public static final WeekDay Friday = new WeekDay();

  private WeekDay() {
  }

  some special helper methods provided for enums (see your text)

  you may add additional methods here
}
```

So, `WeekDay` will be a static "inner" or "nested" class of whatever class you place the enum inside. The `Monday`, `Tuesday`, etc. are fields set to instances of the `WeekDay` class. Because the constructor is private, no other instances can be created than one instance for each of the listed fields. (Note that we do not need to override the `equals` method in an `enum`. Since no other instances can be created than those stored in the fields, you can use == to compare enum values.) As entered, you have a default private constructor for the enum, but you can create your own constructor if you wish. For example, we could create a constructor that takes a `String` that is the name of the day. If we do that, we would need to do the following:

```
enum WeekDay {
  Monday("Monday"), Tuesday("Tuesday"), Wednesday("Wednesday"), Thursday("Thursday"),
Friday("Friday");

  private String name;

  private WeekDay(String name) {
```

```
    this.name = name;
  }
  you may add additional methods here
}
```

What You Need To Do

You will program an almost complete game of chess.

Part 1: ChessPiece

Before you can compile the provided classes, you must create a `ChessPiece` type. The can be a class, abstract class, or interface. The `ChessPiece` class requires the following methods to get the code to compile:

1. `ChessGame.Side getSide()`: to return which player this piece belongs to
2. `String getLabel()`: returns a short string that can be used to indentify the piece when not using a picture
3. `Object getIcon()`: returns a piece of graphics that is how the piece should be drawn (may be `null` for a basic display)
4. `void setLocation(int row, int column)`: lets the piece know where it currently is on the game board
5. `boolean isLegalMove(int toRow, int toColumn)`: returns if it is legal to move the piece from its current location to input location

In addition, every chess piece is required to have to following methods to help you implement the game. Whether you place them in `ChessPiece` or in different types is up to you:

1. `ChessBoard getChessBoard()` returns the chess board that this piece is on.
2. `int getRow()` returns the current row the piece is on.
3. `int getColumn()` returns the current column the piece is on.
4. `boolean isLegalNonCaptureMove(int row, int column)` returns whether it is legal to move this piece from its current position to the indicated position, assuming there is no piece currently at that location.
5. `boolean isLegalCaptureMove(int row, int column)` returns whether it is legal to move this piece from its current position to the indicated position, assuming there is a piece from the other player currently at the destination location.
6. `void moveDone()`: handles the processing (if any) that is needed once the move is completed.

Part 2: Create the individual chess pieces.

Design Rules: Your project must contain a separate class for each of the different types of pieces in the game. You should organize the classes for the pieces into a hierarchy. You *may* use any combination of classes, abstract classes, or interfaces that you feel is appropriate. You *may* add additional types to the ones listed. The classes/types *may* contain additional methods to the ones listed if you feel they are needed. You *may* use any combination of inheritance, method overriding, and method overloading to achieve the needed behavior. Part of the coding grade will be the quality of the hierarchy you create.

Hint: Spend a lot of time designing your hierarchy *before* you code. A well designed hierarchy will reduce the amount of code you have to write.

Hint: Make your hierarchy capture the general types of pieces. For example, both the queen and the bishop may move diagonally. Each piece should not separately define the rules for a diagonal move. Instead, use inheritance so you can define the rule once, and try to make it "universal" so if we want to add a new kind of piece that allows diagonal moves (which we may in a future project), we can do so easily.

Hint: Your piece hierarchy should not contain any Java swing code. All Java swing code should be in the classes doing

the board display. If you want to create icons for your pieces (not required), make separate classes that creates the icons, and have the piece classes store the icon data as Object. This will make it easier to replace Java swing with a different graphics library (that we may do in a future project) and not have to rewrite your piece hierarchy.

1. `KnightPiece`: A knight's legal move is L-shaped: up one space and over two or up two spaces and over one. The spaces the knight moves past are allowed to contain other pieces of either player, but the space the knight ends on must either be empty or contain a piece of the opponent. If it has an opponent piece, the knight captures that piece removing it from the game. You may use "N" as the label for the kNight.

2. `RookPiece`: A rook can move any number of spaces either horzontally or vertically. There must be no piece on a square between the rook's starting and terminating spaces. The space the rook ends its move in must either be empty or contain an opponent's piece. If there is an opponent's piece on the ending square, the rook captures the piece removing it from the game. You may use "R" as the label for the Rook.

3. `BishopPiece`: A bishop can move any number of spaces in a straight diagonal. There must be no piece on a square between the bishop's starting and terminating spaces. The space the bishop ends its move in must either be empty or contain an opponent's piece. If there is an opponent's piece on the ending square, the bishop captures the piece removing it from the game. You may use "B" as the label for the Bishop.

4. `QueenPiece`: A queen can move any number of spaces in a straight line, either horizontally, vertically, or diagonally. There must not be a piece on a square between the queen's starting and terminating spaces. The space the queen ends its move in must either be empty or contain an opponent's piece. You may use "Q" as the label for the Queen.

5. `KingPiece`: The king can move one space in any direction (including diagonal). The space the king ends its move in must either be empty or contain an opponent's piece. If there is a piece of the opponent on that space, the king captures it, removing it from the game. In addition, the king has a special castle move. In the castle move, the king is moved two spaces to the left or right. The castle move is only legal if the king has never moved, there is a rook belonging to the same player in the corner the king is moving toward and the rook has never moved, there are no pieces between the king and the rook, and none of the squares between the king and the rook are threatened. The move is completed by moving the rook to the space between the king's start and destination square. (Optional: prevent a King from moving into check.) You may use "K" as the label for the King.

6. `PawnPiece`: A pawn may move one space in the forwrd direction (directly away from its player's side and toward the opponent's side) and only if there is no piece of either player in the destination space. if there is no piece on the terminating square. If this is the pawn's first move, the pawn may move two spaces in the forward direction as long as there are no pieces on the destination space or the space being skipped. The pawn may move one space diagonally foward only if there is a piece of the opponent's on that destination space. In that case, the pawn captures the opponent piece removing it from the game. Finally, if the pawn lands on last row on the opponent's side of the board, the pawn can be upgraded to another piece (but not a king). I suggest using the `JOptionPane.showInputDialog` to request the desired piece from the user. (Optional: implement the pawn en passant.) You may use "P" as the label for the Pawn.

Part 3: Create a `EuropeanChess` class.

This class should implement the `ChessRules` interface. This class should create the necessary funtions to complete the game. At minimum, the `legalPieceToPlay` should only allow pieces of the player whose turn it is to be selected, and the `nextMove` method should check if the move is legal (by calling the piece's `isLegalMove` method), and if it is, it should move the necessary pieces on the chessboard and change which player the next one to play.

Optional: Creating the check or checkmate rules.

Part 4: Create a `main` method that launches the game.

You need to create a `EuropeanChess` and `EuropeanChessDisplay` instances, create a `ChessBoard`, and then create the necessary pieces and add them to their starting positions on the board. Once you do that, the chessboard should take over and you can start playing the game.