CHESS MODULE REQUIREMENTS
PRO180 – JAVA I PROJECT

OVERVIEW

For this project you will write a chess application. I have broken down the requirements for the application into a number of different modules which you may complete at your own pace and in any order. The order in which they appear in this document is the suggested progression through the modules. Most modules, however, are independent of other modules (with a few notable exceptions) to give you the most freedom in completing the project in your own way and using your own timeline (with some restrictions).

Given that you are in control of this project, **use your time wisely**. This is a very large program which will take a majority of the allotted time. This document contains a section describing a suggested module completion schedule. Take the schedule seriously as it will help you minimize sleepless nights and countless hours of swearing at your laptop screen.

The next section describes the process we will use for passing off modules, followed by how code reviews work, then a section with the suggested calendar of completion. The remaining sections describe each of the modules along with their requirements and any known dependencies.

MODULE COMPLETION PROCEDURE

You will be required to commit your code each day with appropriate comments to the SVN server provided in class. This will not only help you figure out what you have done for the day, but will provide a backup in case your computer crashes. In addition, SVN provides the means to revert your code to a previous state in case you completely hose your project.

Once you have completed your implementation of the requirements for a particular module, you will be assigned a peer reviewer who will look over your code design and style. When the reviewer feels your code passes the white glove tests (described later), then **the two of you** may approach the instructor, who will then spot-check your code by pulling it down from the SVN server. Once the three individuals involved (you, your reviewer, and the instructor) agree that the code is reasonable, you have completed the module. You may not pass off more than one module per class day, although you may perform multiple reviews in a single day.

CODE REVIEWS

THE CONCEPT

During this project you will get the opportunity to review one of your peer's solutions for each module. This peer review allows both the reviewer and the reviewed to learn from each other.

As a reviewer you will be able to:

- See another programmer's style and incorporate things you like into your own coding.
- Improve your ability to look through code written by someone else and make sense of it.
- Increase your understanding of good programming practices, good variable names, etc.

Being reviewed gives you the opportunity to:

- Learn from a peer.
- Improve your coding practices.

GRADING THE REVIEW

Since it is such a valuable experience to be a reviewer and to be reviewed, it is a nontrivial part of your grade. Since you can't pass off your module without being reviewed, you are implicitly graded on the review. As a reviewer you will be graded on how well you examine your peer's code. If the instructor finds things from the

white glove test after you have reviewed the code, you will be docked. The amount you will be docked depends on the severity of the defect.

WHITE GLOVE TESTS

In programming, there are many ways to approach any given problem. Some ways of coding a particular solution are better than others. This is one reason Neumont exists, to help students understand the best practices in programming and design rather than just learn one way to solve the problem. Things like code structure and efficiency are two things to think about when developing a solution. The "white glove tests" described here will hopefully help you avoid some of the bad coding practices that are so common on The Daily WTF. In some degree, your expertise will be determined not only by what you program, but *HOW* you program.

This is not a comprehensive list of java programming sins, but here are a few common "mistakes" that programming noobs make. You clearly don't want to be labeled a noob when you graduate, so we'll give you a head start here.

Make sure inheritance relationships make sense.

```
public class Bishop extends Queen
```

A Bishop is a Queen? Not really. Perhaps a base class that both extend would be better.

Use constructors wisely.

```
public class ChessPiece {
        protected String name;
        protected Color team;

        public ChessPiece() {
                name = "";
        }
```

How about passing the name and the team through the constructor instead of having each subclass set the data members?

Avoid parallel arrays.

```
public class GamePlay {
        private ChessPiece[][] chessSet;
        private String[][] chessboard;
        private Color[][] background;
```

There is a missing abstraction here. How about creating another class that holds all three things? Then you only need one array (2D array here).

Avoid using arrays in the place of classes.

moveReverse(new int[] { i, j }, blockMove);

This is probably a coordinate of some sort, so use java.awt.Point rather than an array.

Avoid using break, continue, and multiple return statements.

There are a lot of examples of this around, just think about another way around the problem. The reason is that break, continue, and multiple returns makes your code harder to understand and harder to debug.

Avoid confusing code (code that NEEDS a comment to make it understandable).

```
public static boolean checkCoor(int x, int y)
{
        if(!board[y-1][x-1].isEmpty()&& !(board[y-1][x-1].equals(""))&&board[y-1][x-1]!=null){
                return true;
        }
        return false;
}
```

Huh? There are actually quite a few problems with this bit of code (including a possible null pointer exception). But what does it do? If nothing else, it creates a nightmare for the person who has to maintain your code after you get fired.

Use meaningful variable and class names.

public class Part1 {

What does the class do? No clue. Using good class names helps you conceptualize the design. Using good variable names is a form of documentation. If you have a good variable name, then maintenance and debugging is easier.

Don't use "Magic Numbers", use constants.

```
this.surface = new Square[8][8];

//fill the board with squares
for(int i = 0; i < 8; i++){
      for(int j = 0; j < 8; j++){
            this.surface[i][j] = new Square(this,new Point(i,j));
      }
}
```

Why not have a constant for BOARD_SIZE or BOARD_ROWS and BOARD_COLUMNS. Either is more descriptive than 8. Either way, what if you accidentally type 88 or 7 instead of 8. Using magic numbers can introduce some really weird bugs into your code.

Use Polymorphism to your advantage.

```
/**
 * Method to determine which "getMoves..." method to call based on piece type.
 * @param col
 * @param row
 * @return ArrayList<String>
 */
public ArrayList<String> getMoves(int col, int row)
{
      ArrayList<String> possibleMoves = new ArrayList<String>();

      switch(pieces[row][col].getPieceType())
      {
      case 'K':
            possibleMoves = getMovesForKing(row, col);
            break;
      case 'Q':
            possibleMoves = getMovesForQueen(row, col);
            break;
      case 'B':
            possibleMoves = getMovesForBishop(row, col);
```

```
                break;
...
```

In the Piece class, there should be a method, getMoves(row,col) that all subclasses must implement. That way, you don't even have to worry about what type of piece it is, just call the method and the right thing will happen.

Avoid duplicate code.

```
if(black)
{
        boardColor[i][j]=Color.RED;
        black=!black;
}
else if(!black)
{
        boardColor[i][j]=Color.RED;
        black=!black;
}
```

This coder fell victim to the classic copy/paste bug. Consider this instead:

boardColor[i][j] == black ? Color.BLACK : Color.RED;
black = !black;

Use switch statements sparingly.

```
private int chooseRowGP(int chooseRow){
      switch(chooseRow){
      case 1:
            return 0;
      case 2:
            return 1;
      case 3:
            return 2;
      case 4:
            return 3;
      case 5:
            return 4;
      case 6:
            return 5;
      case 7:
            return 6;
      case 8:
            return 7;
      }
      return -1;
}
```

Think about this one for a minute. There's a much simpler solution. In general, think twice before using a switch statement. Is there a better way?

If you override equals, ALWAYS override hashCode.

Tons of APIs use both equals and hashCode and you might not even realize it. This can cause some really cryptic errors and may not always be consistent. The APIs depend on the fact that if two objects are equal, they have the same hashCode, so don't violate that contract.

SUGGESTED COMPLETION CALENDAR

MODULE	RECOMMENDED COMPLETION DATE
1: I/O *(Java Basics, I/O, Regex, String Manipulation)*	May 15th
2: Board Display *(Arrays, Collections, Objects, Classes, Inheritance, Abstract Classes, Interfaces, Constructors)*	May 17th
3: Piece Movement *(Exception Handling, Polymorphism, Encapsulation, Access Modifiers)*	May 20th
4: Turn Taking *(Exception Handling, Polymorphism, Encapsulation, Access Modifiers)*	May 22nd
5: User Play *(User Interface)*	May 24th
6: Check *(Recursion)*	May 28th
7: Checkmate *(Recursion)*	May 30th
8: GUI *(Swing, Events, Threading, Anonymous Classes)*	June 3rd
Final Deadline for all modules	June 4th

Although you are not required to pass off on the recommended date, remember that you may not submit more than one module per class day. No modules may be submitted after the final deadline on June 4th.

FILE I/O MODULE

DEPENDENCIES: None

DESCRIPTION

For the File I/O module, you must be able to read in a set of inputs from a stream, then process them one at a time, and finally spit them out in another format. You will be parsing moves from a file with the file name given to you at the command line.

The input will have one move per line. That move may include:

- Placing a new piece at a specific location.

  Examples:

  - qld1 // places a white (light) queen on d1

- kde8 // places a black (dark) king on e8

- Moving a single piece from a location to another location.

  Examples:

  - d8 h4 //moves the piece on d8 to h4
  - c4 d6* //moves the piece on c4 to d6 and captures the piece on d6

- Moving two pieces.

  Examples:

  1. e1 g1 h1 f1 // moves the king from e1 to g1 and the rook from h1 to f1 (called king-side castling)

Your output should indicate what the move was or indicate that the move was invalid if it isn't in one of the listed formats.

The reviewers will focus on the correct use of classes in the java.lang package, the java IO framework, and regular expressions in addition to the white glove tests detailed previously.

BOARD DISPLAY MODULE

DEPENDENCIES: None

DESCRIPTION

For the board display module, you must be able to set up a chess board with the standard 32 pieces and display it.

The reviewers will focus on the correct use of arrays, collections, and general object-oriented programming practices in addition to the white glove tests detailed previously.

PIECE MOVEMENT MODULE

DEPENDENCIES: File I/O Module, Board Display Module

DESCRIPTION

For the piece movement module, you must be able to read in a set of piece placements and moves from a file, modify your board appropriately, and display the updated board after each move.

If any move is invalid according to the standard chess rules (e.g. a bishop not moving diagonally or a knight going in a straight line), display an error message, cancel the move, and continue parsing the file. You do not need to worry about turn taking, pawn promotion, capturing en passant, or castling for this module.

The reviewers will focus on the correct use of exception handling and polymorphism in addition to the white glove tests detailed previously.

TURN TAKING MODULE

DEPENDENCIES: File I/O Module, Board Display Module

DESCRIPTION

For the turn taking module, you must be able to read in a set of moves from a file, modify your board appropriately, and display the updated board after each move.

If the play does not alternate between white and black moves, display an error message, cancel the move, and continue parsing the file. For example, if the file indicates white, black, white, white, black, you would skip the 4th command (as it was black's turn).

If any move is invalid according to the standard chess rules, display an error message, cancel the move, and continue parsing the file. You do not need to worry about pawn promotion, capturing en passant, or castling for this module.

The reviewers will focus on the correct use of exception handling and polymorphism in addition to the white glove tests detailed previously.

USER PLAY MODULE

DEPENDENCIES: Board Display Module, Piece Movement Module, Turn Taking Module

DESCRIPTION

For the user play module, you allow input from the user rather than from a file. For each player's turn:

1. Show pieces with valid moves and allow the user to select one of those pieces.
2. Show all valid moves for the selected piece and allow the user to select one of the available moves.

Do not allow the user to select an invalid piece or an invalid move. If they do, display a message indicating the selection was invalid and to try again.

The reviewers will focus on the user interface in addition to the white glove tests detailed previously.

CHECK MODULE

DEPENDENCIES: File I/O Module, Board Display Module

DESCRIPTION

For the check module, you must be able to read in a set of piece placement moves from a file to construct a game board. Once the board is constructed, display it.

Assume first that it is white's turn and determine whether or not white is in check and display to the user your findings. Then assume it is black's turn and determine whether or not black is in check and display to the user your findings.

The reviewers will focus on the white glove tests detailed previously.

CHECKMATE MODULE

DEPENDENCIES: File I/O Module, Board Display Module

DESCRIPTION

For the checkmate module, you must be able to read in a set of piece placement moves from a file to construct a game board. Once the board is constructed, display it.

Assume first that it is white's turn and determine whether or not white is in checkmate and display to the user your findings. Then assume it is black's turn and determine whether or not black is in checkmate and display to the user your findings.

The reviewers will focus on the white glove tests detailed previously.

GUI MODULE

DEPENDENCIES: User Play Module

DESCRIPTION

For the GUI module, you must be able to display a board using swing, select a valid piece using the mouse, display valid moves, and then allow the user to select a destination. You should then output to the console the user's move in the standard format described in the File I/O module.

The reviewers will focus on the correct use of swing, events, and anonymous inner classes in addition to the white glove tests detailed previously.