Assignment 4/6: The chess game

Object-Oriented Programming and Design B2-2011

Department of Computer Science at the University of Copenhagen (DIKU)

Deadline: 2011-12-17, 18:00 CET

Preface

This is one of the six weekly assignments in the course. Every assignment gets a score of either 0, 0.5, or 1, where 1 is full credit, and 0 is no credit. You are required to acquire a sum of at least 4 points (out of the possible 6) to be admitted to the exam.

Types of exercises

The assignments themselves consist of two types of exercises, those that require you to do coding only, and those that require you to respond in prose, or by taking a screenshot. Exercises that require you to do things other than coding are always postfixed with a *.

What we expect of your code

For the coding exercises we expect of you to follow the Code Conventions for the Java Programming Language[1]. Deviation from the conventions does not necessarily lead to a reduced score, but the conventions are in place to ensure that your instructor can read your source code quickly. Additionally, make sure to follow the explicit code specifications in the assignment text as your code might be subject to automated tests. Such tests usually rely on various classes, fields, methods, etc. to be named in a specific way. Also, make sure to structure your code in an intuitive way, and document your source code and interfaces where appropriate. What this last statement means exactly should become ever more clear as we advance through the course.

What to hand in

For each assignment you should hand in *one* zip archive. The actual structure of the archive, and hence the files we expect you to package into it, are specified at the end of every assignment. To allow us to easier grade the assignments, you are strictly required to follow the following naming convention:

`<full-name>-A.<number>.zip`

The name must additionally be lower-cased. `<full-name>` should be a dash-separated string of characters `a` to `z` representing your name as registered on Absalon. If you have special characters in your name, such as `æ`, `ø` and `å`, please use their `a` to `z` equivalents of `ae`, `oe` and `aa`, respectively. So for example, if your name is Kåre Søren Æbelsen, for the first assignment we'd expect you to hand in a file with the following name:

`kaare-soeren-aebelsen-A.1.zip`

Please make sure that you hand in your *source code* and *not* java bytecode. It is always a good idea to check that all of your files have been uploaded correctly by downloading them again and browsing them yourself.

As for the written exercises, your solutions should be written in plain text format. We make no explicit requirements as to whether they should be written in Danish or English, but try to stick to one. Additionally, this depends on the preferences of your instructor.

[1] http://www.oracle.com/technetwork/java/codeconv-138413.html

Collaboration and help

We encourage you to discuss your solutions with your peers, but your solution *must* be individual. We expect you not to share explicit code or to take credit for another student's work. All in all, we expect you to honor each other's, and your own work.

You may ask your instructor for general help with the assignment or for additional clarifications. You can also use the course discussion forum on Absalon for the exact same purposes. Please remember to read the forum rules before you post anything on the forum. You should especially remember to prefix the forum topics with e.g. A.1.2 if you're asking a question about exercise 2 in assignment 1.

Resubmissions

If you were so unfortunate to not get full credit for your solution on the first try, you can revise, and resubmit your solution. This goes for the first 5 the first time you submit assignments, but *not* the 6th due to time constraints.

The resubmitted file should follow the following naming convention (continuing our running example):

```
kaare-soeren-aebelsen-A.1.resubmit.zip
```

Deadlines

The initial assignment deadline is usually **Saturday, 18:00 CET**. The instructors should have graded your assignment before the following **Wednesday, 23:59 CET**. The deadline for the resubmission is hence the immediate **Sunday, 23:59 CET**. This gives you two full work days to work on you resubmission. You may have a special agreement with your instructor regarding these deadlines in general.

Some conventions

When we say "book", we refer to the main course literature, namely Objects First With Java 4th Edition, by David Barnes & Michael Kölling, and when we say "project library" we refer to the project library on the accompanying CD, also provided via Absalon under "Course material".

Handed out files

```
./A.4.pdf
./src/Piece.java
./src/Position.java
./test/PositionTests.java
```

Credit assignment

The intent of this assignment is for you to exercise your understanding of inheritance, sub typing and polymorph methods. You are building a simple chess game. The assignment is divided into thee parts, the chess board, the chess piece, and the player interaction.

A.4.1 The pieces, part one

We'll represent the individual types of pieces (rook, knight, bishop, etc.) using a class hierarchy. There are however some common properties to all pieces on a chess board, at the very least, their color. We've provided a starting point for the `Piece` class, and would like for you to complete the implementation.

1. Define an *enum type* `Color` within the class `Piece` that is either `BLACK` or `WHITE`.

   Hint: Enums are coverd in the book in § 7.13. Refer to http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html for further reference.

2. Define a constructor for the class `Piece`; note, that this constructor can only be called by nested classes. The color of the piece should be specified when the class is instantiated, and shouldn't change throughout the object's lifetime.

3. Override the method `toString` of the `Piece` class to return the 2-character representation of the piece. White pieces are prefixed by the character `W`, and black pieces are prefixed by the character `B`. All pieces are postfixed by their *key*, that is, either `P`, `R`, `N`, `B`, `Q`, or `K`. For instance, the black king should be represented by the string `BK`.

A.4.2 Positions

Complete the definition of the handed out `Position` class. We've provided quite a bit of the implementation. The most notable is `public static Position fromRepresentation(String representation)`, which is the only way that other objects can construct positions. The method takes in a `String` representing a position in regular chess notation[2], so e.g. passing `A2` yields the position where $x = 0$ and $y = 1$.

Note: Invalid positions (outside the bounds of the board) cannot be created.

The remaining methods, namely `public boolean equals(Object other)` and `public int hashCode()` might be of little interest to you, but they allow you to compare positions for equality with ease.

1. Define the accessor methods `getX()` and `getY()`.

By the time you're done with this, you should passing all tests in the handed out `test.PositionTests`.

A.4.3 The board

Create a class `Board` to represent the chess board. A chess board keeps track of a two-dimensional matrix of `Piece` objects where each entry is a field on the chess board. The matrix can be represented using a nested array, namely `Piece[][]`. The matrix should be initialized in the constructor to be 8×8 in size. Think of the index $[0][0]$ in the array as the lover left corner of the chess board (1A) and $[7][7]$ as the upper right corner (8H).

A.4.4 Who's there?

Implement the following method in the `Board` class:

```
1 public Piece getPiece(Position position)
```

It should return either `null` or an instance of a `Piece`, all depending on what's at the specified location.

A.4.5 The pieces, part two

Implement the following method in the `Piece` class:

```
1 public boolean isLegalMove(
2   Position start,
3   Position end,
4   Board board)
```

Note: You can assume that the start and end positions are within the bounds of the board.

The method should return `true` if the piece at the end position is either `null` or a piece of a different color than the current instance, and `false` otherwise.

In descending types we'll override this functionality, and add additional logic to test whether the laws of chess actually allow us to make this sort of move with the current type of chess piece.

[2] Refer to Figure 1 for an explanation of the notation.

A.4.6 Something in the way

Implement the following method in the `Board` class:

```
1 public boolean allPositionsAreEmpty(Position[] positions) { }
```

It should return `true` if and only if all the specified positions contain the value `null`, and `false` otherwise.

A.4.7 More pieces

In the following exercise you're going to define the more concrete pieces of a chess game. Only the pawn and king pieces are *mandatory*, all other pieces are *optional*.

1. Define six descendants to the `Piece` class to represent the six different types of pieces in a chess game, namely `King`, `Queen`, `Bishop`, `Knight`, `Rook` and `Pawn`.

   Note: Make sure to call the appropriate superclass constructor in every constructor of the new classes. If you're in doubt how this is done, refer to http://docs.oracle.com/javase/tutorial/java/IandI/super.html.

2. For each class, *override* the method `isLegalMove`, such that, in addition to checking for the end position to be empty or an enemy piece, it also checks to see if the move is allowed for the current piece according to the laws of chess. We've summarised the laws below, for more information you can visit http://en.wikipedia.org/wiki/Rules_of_chess.

   (a) No pieces go off the board, that is, they are bound by the 8×8 square.

   (b) No pieces can make a move to a location already occupied by a piece of the same color.

   (c) Most pieces (except knights) cannot jump over other pieces (of any color), when making a move, the path from start to end has to be completely clear.

   (d) Most pieces (except pawns) capture by performing a legal (for them) move into the position occupied by a piece of a different color.

   (e) The king can move exactly one square horizontally, vertically, or diagonally.

   (f) The rook moves any number of vacant squares vertically or horizontally.

   (g) The bishop moves any number of vacant squares in any diagonal direction.

   (h) The queen can move any number of vacant squares diagonally, horizontally, or vertically.

   (i) The knight moves to the nearest square not on the same rank, file, or diagonal. In other words, the knight moves two squares horizontally then one square vertically, or one square horizontally then two squares vertically. Its move is not blocked by other pieces, that is, it jumps to the new location.

   (j) Pawns have the most complex rules of movement:

      i. A pawn can move forward one square, if that square is unoccupied.
      Note: Disregard the usual ability of the pawn to move *two squares forward* if it is in its *initial position*.

      ii. A pawn cannot move backwards, that is, away from the opponent's side of the board.

      iii. Pawns are the only pieces that capture differently from how they move. They can capture an enemy piece on either of the two spaces adjacent to the space in front of them (i.e., the two squares diagonally in front of them) but cannot move to these spaces if they are vacant.

A.4.8 Setting up the board

Implement the following method in the `Board` class:

```
1 private void initialize() { }
```

Make sure to call it from the `Board` constructor.

The method should layout the board with pieces as in a conventional chess game. To avoid ambiguities, Figure 1 showcases what an initialized board looks like.

Note: Since not all pieces are mandatory, ignore the piece you haven't created subclasses for, that is, don't place them on the board.

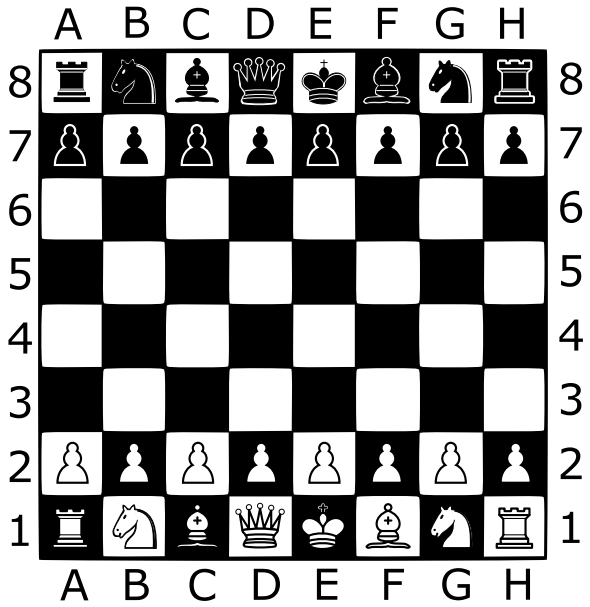

Figure 1: The initial layout of a chess game. The top row: rook, knight, bishop, queen, king, bishop knight, rook. The pieces along the second row from the top are all pawns. This figure should also serve as an explanation of *regular chess notation*, where e.g. the black rook is located at A8.

A.4.9 A human-readable state of the board.

Implement the following method in the `Board` class:

```
1 public String humanReadableState() { }
```

For an initial setup, the method should produce the following string:

```
BR BN BB BQ BK BB BN BR
BP BP BP BP BP BP BP BP
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
WP WP WP WP WP WP WP WP
WR WN WB WQ WK WB WN WR
```

Hint: Remember that we've already defined the `toString()` method for all pieces to return the representation of a piece in exactly this type of notation.

Note: `String` concatenation (+) is a terribly inefficient way to construct long `Strings`. This is because every time you have a line of code that uses +, a new `String` is generated. A new `String` is generated

by copying the contents of the operators into a new memory location. If you now imagine a loop that continually builds up a `String`, it should be clear to you how much extra work is done in every iteration.

This leads to the general rule of thumb that whenever you have a loop to build a string (in this case, we're looping over the matrix representing the board), you should use a `StringBuilder` (or something alike). The class is just like an `ArrayList` that you add elements to, and then call the method `toString()` to finally combine all the constituents into a single `String`. The following pseudocode shows you the most important (to you) methods of the `java.lang.StringBuilder` class:

```
1 StringBuilder builder = new StringBuilder();
2 for (...)
3 {
4   builder.append(...);
5 }
6 String result = builder.toString();
```

A.4.10 Captured king

In general we won't keep track of captured pieces for simplicity, although you may implement this if you wish. The most important thing is for the board to note when either the white or black king is captured, since this means *game over*.

Implement some public accessor method that allows outsiders to check if the game is over or not, and hence also allows to determine the color that won the game.

A.4.11 Move

Implement the following method in the `Board` class:

```
1 public boolean tryMovePiece(Color playerColor, Position start, Position end) {
    }
```

The return value should indicate if the move succeeded or not.

- If the game is over, the move fails.
- If the start position does not refer to an instance of the `Piece` class, the move fails.
- If the piece located at the start position does not have the same color as `playerColor`, the move fails.
- If the move is not legal for the given `Piece`, the move fails.
- Otherwise, the move succeeds, and the starting position is cleared, while the end position is filled with the `Piece` originally located at the start position.

  Note: You should probably check to see if the figure in the end position is a king, and declare the game over if it is captured.

A.4.12 The game

Create the class `Game` that represents the actual chess game. The class should initialize the board and two players, and keep track of their turns.

It is up to you how to represent the players, but for the sake of simplicity we'll *let player 1 always be white*, and *player 2 always be black*. Due to a convention used in chess, this superimplies that *player 1 always makes the first move*.

1. Implement the following method:

```
1 public void restart() { }
```

The method should reset the board and the players.

2. Implement the following method:

```
1 public boolean tryMovePiece(Position start, Position end) { }
```

All depending on who's turn it is, the method should attempt to move the piece from `start` to `end` on the underlying board, while passing the color of the *current* player.

Much like the corresponding method on the board, the return value should indicate if the move was successful or not.

A.4.13 The command-line interface

Create the class `Main` that interfaces with `System.out` and `System.in` to run the game, that is, this is a console game.

Each time the game starts, show the state of the initial board.

The players should take turns entering moves in regular chess notation, refer to Figure 1 if you're in doubt what each position is called. The first argument is the (possibly empty) start position, and the second argument is the desired end position. The arguments are *always* separated by a single space. The positions, as well as the move, may or may not be legal. Additionally, we have the following commands:

- Either player may at any time issue the command `board` to get a human readable view of the board.
- Either player may at any time issue the command `handshake` to indicate a negotiated draw and restart the game. For user-friendliness, show the board after the game has been restarted.

If some player manages to capture the other player's king (A.4.10), the game is over. The winner should be declared, and the game restarted.

Here's a sample session, remember that lines starting > are those put into the program by a user:

```
BR BN BB BQ BK BB BN BR
BP BP BP BP BP BP BP BP
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
WP WP WP WP WP WP WP WP
WR WN WB WQ WK WB WN WR
Player 1:
> e2 e3
ILLEGAL!
Player 1:
> E2 E3
Player 2:
> E7 E6
Player 1:
> E3 E5
ILLEGAL!
Player 1:
> handshake
BR BN BB BQ BK BB BN BR
BP BP BP BP BP BP BP BP
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
WP WP WP WP WP WP WP WP
WR WN WB WQ WK WB WN WR
Player 1:
> E2 E3
Player 2:
> G8 F6
Player 1:
> E3 F6
ILLEGAL!
Player 1:
> board
BR BN BB BQ BK BB 00 BR
BP BP BP BP BP BP BP BP
00 00 00 00 00 BN 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 WP 00 00 00
WP WP WP WP 00 WP WP WP
WR WN WB WQ WK WB WN WR
Player 1:
> E3 E4
Player 2:
> F6 E4
Player 1:
> board
BR BN BB BQ BK BB 00 BR
BP BP BP BP BP BP BP BP
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 BN 00 00 00
00 00 00 00 00 00 00 00
WP WP WP WP 00 WP WP WP
WR WN WB WQ WK WB WN WR
```

Files expected to be handed in

We expect for *at least* the following list of files to be handed in for full credit. `./` resembles the root of your zip archive.

```
./chess/Position.java
./chess/Piece.java
./chess/Board.java
./chess/King.java
./chess/Pawn.java
./chess/Game.java
./chess/Main.java
```

Challenge of the week

Challenge of the week is an optional part of the weekly assignment for students that have completed the mandatory exercises above. You should feel confident enough about your solutions to the exercises

above before taking up the challenge, as there is *no extra credit* for solving it. However, the most elegant solution is rewarded recognition and a free beverage from the canteen. The elegance of the solution is judged by it's exploitation of object-orientation, the Java language and it's libraries.

The deadline for completing the challenge is the same as the assignment deadline, however there are *no* resubmissions. You should upload your solution to the challenge into a separate folder, which you'll find on Absalon.

The challenge

Let the players save and restore the state of a given game using the file system. Consider the most space-efficient (in terms of hard-disk space) way to do so. The state of the game is both the positions of the pieces, and the moves the players have made so far. In so far as this is possible, it should also be possible to *reverse* any game.