**BoardLayout**

**com.ipl.training.induction.draughts.model.BoardLayout**

**Class Specification**

**Project Reference 999/170**

**Document Reference BoardLayout\_cs**

**24 September 2013**

**Issue 9**

**Copyright Notice**

This document has been prepared by IPL Information Processing Limited (“IPL”) and, subject to any existing rights of other parties, IPL is the owner of the copyright of this document. No part of this document may be copied, reproduced, stored in a retrieval system, disclosed to a third party or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of IPL.

**© IPL Information Processing Limited 2013**

**Document History**

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Issue | Author | Comments |
| 10/06/2010 | 1 | Brian Bassil | Issued |
| 16/09/2010 | 2 | Christine Harrison | Corrections after first run of course |
| 22/10/10 | 3 | Christine Harrison | More corrections after second run of course |
| 08/11/2010 | 4 | Christine Harrison | Added IllegalArgumentException in the ‘Throws’ field of processFen method. |
| 24/11/2011 | 5 | Phil Lewis, C Harrison | Updated to reflect new Java standard, including diagrams. |
| 16/10/2012 | 6 | Andrew Jenkins | Removed final modifier from public methods, 5.1, class is final so this was redundant.  Corrected Typo in 5.4.3 input table.  Changed confusing local identifier name in 5.2.2. |
| 13/06/2013 | 7 | C Harrison | Updated some processing logic to explain what’s going on. |
| 20/09/2013 | 8 | C Harrison | Corrected determineMove() processing logic |
| 24/09/2013 | 9 | C Harrison | Really corrected determinMove() processing logic this time. |

Contents

[1 Package Identity 1](#_Toc358905044)

[2 Description 1](#_Toc358905045)

[3 Interface 1](#_Toc358905046)

[4 Structure 2](#_Toc358905047)

[5 Element Descriptions 3](#_Toc358905048)

[5.1 Public Methods 3](#_Toc358905049)

[5.1.1 equals 3](#_Toc358905050)

[5.1.2 hashCode 3](#_Toc358905051)

[5.1.3 iterator 4](#_Toc358905052)

[5.1.4 toString 4](#_Toc358905053)

[5.2 Package Access Methods 6](#_Toc358905054)

[5.2.1 BoardLayout(String) 6](#_Toc358905055)

[5.2.2 BoardLayout(BoardLayout, int, int) 6](#_Toc358905056)

[5.2.3 canSelect 8](#_Toc358905057)

[5.2.4 diff 9](#_Toc358905058)

[5.2.5 getCaptured 10](#_Toc358905059)

[5.2.6 getCurrentPlayer 10](#_Toc358905060)

[5.2.7 getGameState 10](#_Toc358905061)

[5.2.8 getMoves 11](#_Toc358905062)

[5.2.9 getRank 12](#_Toc358905063)

[5.2.10 getSquare 12](#_Toc358905064)

[5.2.11 isEdge 13](#_Toc358905065)

[5.3 Protected Methods 13](#_Toc358905066)

[5.4 Private Methods 14](#_Toc358905067)

[5.4.1 canCapture 14](#_Toc358905068)

[5.4.2 determineAllMoves 14](#_Toc358905069)

[5.4.3 determineMove 16](#_Toc358905070)

[5.4.4 initialBoardLayout 17](#_Toc358905071)

[5.4.5 next 18](#_Toc358905072)

[5.4.6 processFEN 18](#_Toc358905073)

[5.4.7 setSquare 19](#_Toc358905074)

[5.5 Public Fields 20](#_Toc358905075)

[5.6 Package Access Fields 20](#_Toc358905076)

[5.7 Protected Fields 20](#_Toc358905077)

[5.8 Private Fields 20](#_Toc358905078)

[5.9 Public Nested Classes 23](#_Toc358905079)

[5.10 Nested Classes 23](#_Toc358905080)

[5.10.1 InvalidMoveException 23](#_Toc358905081)

[5.10.2 BoardLayoutIterator 24](#_Toc358905082)

[6 Resource Requirements 27](#_Toc358905083)

[7 Test Plan 27](#_Toc358905084)

[7.1 testConstructorWithNoFen 27](#_Toc358905085)

[7.2 testConstructorWithInvalidFen 27](#_Toc358905086)

[7.3 testToString 27](#_Toc358905087)

[7.4 testEquality 27](#_Toc358905088)

[7.5 testHashCode 28](#_Toc358905089)

[7.6 testGetGameState 28](#_Toc358905090)

[7.7 testInvalidMove 28](#_Toc358905091)

[7.8 testValidMove 28](#_Toc358905092)

[7.9 testCapturePiece 28](#_Toc358905093)

[7.10 testDiff 28](#_Toc358905094)

[7.11 testCanSelectOrdinaryMoves 29](#_Toc358905095)

[7.12 testCanSelectCaptureMoves 29](#_Toc358905096)

[7.13 testGetRank 29](#_Toc358905097)

[7.14 testIsEdge 29](#_Toc358905098)

[7.15 test50MoveDraw 29](#_Toc358905099)

[7.16 testPieceResetDrawCounter 29](#_Toc358905100)

[7.17 testCaptureResetDrawCounter 29](#_Toc358905101)

[7.18 testIterator 29](#_Toc358905102)

[7.19 testIteratorUnsupportedOperations 29](#_Toc358905103)

[8 Scenarios 29](#_Toc358905104)

[9 Supporting Documentation 29](#_Toc358905105)

# Package Identity

Class Name - BoardLayout  
Class Identity - com.ipl.training.induction.draughts.model.BoardLayout

# Description

BoardLayout describes the current layout of pieces on the board. The class is immutable, new BoardLayout objects can be created via one of two constructors, one takes a FEN tag to describe the intial layout and the other takes a BoardLayout and two integers describing a move to apply.

BoardLayout contains two nested classes:

* InvalidMoveException which extends Exception and is thrown when an invalid move is attempted.
* BoardLayoutIterator which is a private class that implements Iterator<SquareData> and is constructed by BoardLayout.iterator().

# Interface

|  |  |
| --- | --- |
| Attribute | Value |
| Visibility | Package |
| Modifiers | Final |
| Extends | - |
| Implements | Iterable<SquareData> |

# Structure



# Element Descriptions

## Public Methods

### equals

An overridden version of equals() may be helpful to implementers of AbstractComputerPlayer. If equals is overridden hashCode() must also be overridden with a consistent implementation.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Object.equals() |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| obj | Object | The reference object with which to compare |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| boolean | True if obj is equal to this BoardLayout, otherwise false. |

**Processing Logic**

* If obj is null return false
* If obj is not the same class as **this** return false
* Create a local variable of type BoardLayout called layout and store obj in it. If layout.squares.length is not equal to **this**.squares.length then return false.

Loop over the number of squares and for each iteration check if **this**.squares is not equal to layout.squares. If they aren’t equal then return false.

* If current player from **this** is not equal to the current player from layout then return false.
* Return true.

### hashCode

An overridden version of hashCode() may be helpful to implementers of AbstractComputerPlayer. If equals is overridden equals() must also be overridden with a consistent implementation

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Object.hashCode() |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| int | A hash code value for this object. |

**Processing Logic**

The full implementation details will require testing and tuning to ensure performance. See the javadoc for Object.hashCode() for the contract that equals() must obey. As BoardLayout is immutable caching the generated hash value is a valid optimisation.

### iterator

This method is defined by the Iterable interface. It returns a new BoardLayoutIterator which allows the caller to iterate over the pieces on the board.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Iterable<SquareData>.iterator() |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| Iterator<SquareData> | An iterator that iterates over all the non-empty Squares on the board. |

**Processing Logic**

Construct and return a new BoardLayoutIterator passing in **this**.

### toString

Constructs a FEN tag representing the current board layout.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Object.toString() |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| String | a String containing a FEN tag representing this board layout |

**Processing Logic**

* Create a final local variable called str of type StringBuilder
* If currentPlayer is BLACK
  + Construct a new StringBuilder with “B:” and assign it to str
* Otherwise
  + Construct a new StringBuilder with “W:” and assign it to str
* Create a final local variable whitePieces of type StringBuilder and initialise it with a new StringBuilder created with “W”
* Create a final local variable blackPieces of type StringBuilder and initialise it with a new StringBuilder created with “B”
* Create two booleans called firstWhite and firstBlack and initialise them to true.
* Loop over **this** using sqData as the loop variable
  + Assign the content of sqData to a local variable contents
  + If contents is white
    - If firstWhite is false append “,” to whitePieces
    - Set firstWhite to false
    - If contents is a king append “K” to whitePieces
    - Append the squareID of sqData to whitePieces
  + Otherwise
    - If firstblack is false append “,” to blackPieces
    - Set firstblack to false
    - If contents is a king append “K” to blackPieces
    - Append the squareID of sqData to blackPieces
* If firstWhite is false
  + - append whitePieces to str
    - append “:” to str
* If firstBlack is false
  + - append blackPieces to str
* Return str as a String

## Package Access Methods

### BoardLayout(String)

Constructs a new BoardLayout based on a FEN tag. Throws IllegalArgumentException if the FEN tag contains invalid data.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | IllegalArgumentException |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| fen | String | A String containing a FEN tag that describes the initial board layout. |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* Initialise squares to a new Square[] of length NUMBER\_OF\_SQUARES
* If fen is null or an empty string call intialBoardLayout()
* Otherwise call processFEN()
* Set captured to null
* Set turnCount to zero.
* Call determineAllMoves()

### BoardLayout(BoardLayout, int, int)

Constructs a new BoardLayout based on an existing BoardLayout and a move.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | InvalidMoveException |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| board | BoardLayout | The initial board layout. |
| start | int | The starting point of the move. |
| end | Int | The ending point of the move. |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

Set this board to the initial board layout to begin:

* Copy squares from board using Arrays.copyOf.
* Assign board.getCurrentPlayer() to currentPlayer.
* Initialise blackMoves to a copy of board.blackMoves
* Initialise whiteMoves to a copy of board.whiteMoves

Get the selected square (i.e. ‘start’) and the list of moves this square can make:

* Get the Square at start and store in selectedPiece.
* Create a local variable of type MoveData called possibleMoves and initialise it with getMoves(currentPlayer).get(start).

Now actually change the board layout according the move (i.e. ‘end’)

* Use possibleMoves to check that the Square at start can move to end – if not throw an InvalidMoveException
* Call setSquare(end, selectedPiece);
* Call setSquare(start, Square.EMPTY);

And check to see if it was a capturing move. If it was a capturing move then reset turnCount to help determine when a draw should be declared, remove the first captured piece from the board and see if there are more capturing moves to be made, swapping the current player when there are no further moves:

* Assign the return from possibleMoves.getCapturedPiece(end) to a new local variable capturedSquare.
* If capturedSquare is not 0
  + Set turnCount to 0
  + Create a new SquareData to describe the captured piece and store it in captured.
  + Call setSquare(captured, Square.EMPTY)
  + Call determineAllMoves()
  + Assign the return from getMoves(getCurrentPlayer()).get(end) to a new local variable nextMoves.
  + If nextMoves is null or nextMoves.canMakeCapturingMoves() returns false
    - Call next()

If it wasn’t a capturing piece then swap the current player and set the turn counter to help determine when a draw should be declared:

* Otherwise
  + Set captured to null.
  + Call next()
  + If selectedPiece was not a king
    - set turnCount to zero
  + Otherwise
    - Set turnCount to board.turnCount+1

Next determine if the moved piece should be made into a king:

* If selectedPiece.isKing() returns false
  + If selectedPiece.isBlack()
    - If BoardLayout.getRank(end, PlayerColor.BLACK returns 8
      * Call setSquare(end, Square.BLACK\_KING)
  + Otherwise
    - If BoardLayout.getRank(end, PlayerColor.WHITE returns 8
      * Call setSquare(end, Square.WHITE\_KING

Finally make sure that the blackMoves and whiteMoves fields are updated for the new board layout.

* Call determineAllMoves().

### canSelect

Returns a boolean indicating if the current player can select squareID.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | Integer | The square ID to query |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| boolean | True if the current player can select squareID, otherwise false |

**Processing Logic**

A square can be selected if:

* The square is not empty,
* The piece in the square matches the colour of the player,
* The piece in the square can move.

### diff

Compares two layouts and returns a Set containing the differing squares.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | static |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| oldLayout | BoardLayout | The intial layout |
| newLayout | BoardLayout | The new layout |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| Set<SquareData> | A set of SquareData that describes the changes needed for oldLayout to match newLayout. This will be an empty set if oldLayout matches newLayout. |

**Processing Logic**

* Construct a new HashSet of SquareData.
* If newLayout is null throw an IllegalArgumentException
* If oldLayout is null then all the squares are different so add them all to the HashSet:
  + loop through newLayout.squares creating new SquareData objects and adding them to the hashset
* Otherwise only add the different ones to the HashSet
  + If oldLayout.squares.length != newLayout.squares.length throw an IllegalArgumentException.
  + Loop through the two arrays, comparing each element. If one doesn’t match then create a new SquareData describing the contents in newLayout and add it to the HashSet.
* Return the HashSet.

### getCaptured

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| SquareData | A SquareData describing the piece that was captured creating this layout, or null |

**Processing Logic**

* Return captured.

### getCurrentPlayer

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| PlayerColor | The current player |

**Processing Logic**

return currentPlayer

### getGameState

Returns the state of the game in this layout e.g. GameState.IN\_PROGRESS if the game is still going on.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| GameState | An enum representing the current state of the game |

**Processing Logic**

* Get the possible moves for the current player and assign to a local variable possibleMoves.
* Loop over possibleMoves.values()
  + If any of the values in possibleMoves returns true from canMove() return GameState.IN\_PROGRESS.
* If turnCount equals TURN\_LIMIT
  + Return GameState.DRAW
* If the current player is WHITE
  + return GameState.BLACK\_WIN
* Otherwise
  + Return GameState.WHITE\_WIN

### getMoves

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| player | PlayerColor | The player to get moves for |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| SortedMap<Integer, MoveData> | A map of SquareID to MoveData representing legal moves. |

**Processing Logic**

* If player is PlayerColor.BLACK return a **copy** of blackMoves – otherwise return a **copy** of whiteMoves.

### getRank

Given a square ID and a Player getRank returns the rank. For a black player rank 1 contains squares 1-4 and rank 8 contains squares 29-32. For a white player rank 1 contains squares 29-32 and rank 8 contains squares 1-4. This is done via a lookup table for speed.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | static |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | int | A square ID |
| player | PlayerColor | The player to get the rank of squareID for. |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| int | A number between 1 and 8 inclusive. |

**Processing Logic**

* if (player equals PlayerColor.BLACK)
  + return RANKS[squareID-1]
* else
  + return 9 - RANKS[squareID-1]

### getSquare

Gets a square from squares based on a square ID. (That is, this method converts from 1-based ‘Square ID’ values used elsewhere to 0-based array indexes used internally in BoardLayout).

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | int | The square ID to query |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| Square | The square at squareID |

**Processing Logic**

* Return squares[squareID – 1]

### isEdge

Method to determine if a specific square is an edge square. This is done via a lookup so as to be fast.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | static |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | int | The square ID to query |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| boolean | True if squareID is an edge square, otherwise false. |

**Processing Logic**

* return EDGE\_SQUARES [squareID - 1];

## Protected Methods

None

## Private Methods

### canCapture

Returns a boolean indicating if the specified square can capture in the suggested direction

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | int | The square ID to check |
| direction | Direction | The Direction to check for a capturing move |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| boolean | A boolean indicating if the piece in squareID can make a capturing move in direction. |

**Processing Logic**

Get the contents of the square next to squareID in direction:

* Get the Square at squareID -1 and store in a local variable square.
* Get the NextSquare object for squareID -1.
* Use direction to get the next square integer from the next square object and store it as nextSquareInteger.
* Call squares[nextSquareInteger – 1] and store the result as a local variable called nextSquare.

Now determine if there is a capturing move:

* If the nextSquare is empty return false.
* If square colour is opposite to the nextSquare colour
  + Create a new variable of type Integer called squareAfterNext and store NEXT\_SQUARES[nextSquareInteger -1].get(direction) in it.
  + If squareAfterNext is not zero.
    - If the square at squareAfterNext is empty then return true
* Return false

### determineAllMoves

Determines all of the possible moves with the current board layout. This is done by looping over the all of the squares on the board and all directions and calling determineMove.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* if turnCount is equal to TURN\_LIMIT – there are no moves:
  + clear whiteMoves and blackMoves
  + return.

Now check to see if there are any moves – for each square on the board, check in every direction to see if any move is possible:

* Create four local variables of type TreeMap<Integer, MoveData> named whiteCaptuingMoves, whiteOrdinaryMoves, blackCapturingMoves and blackOrdinaryMoves.
* Loop from 1 to NUMBER\_OF\_SQUARES with a loop counter i
  + Set up a local variable square with getSquare(i).
  + If the square is EMPTY or null then continue the loop.
  + Create two local variables of type TreeSet<Integer> named ordinaryMoves and capturingMoves. These will store the square a piece moves to with an ordinary or capturing move respectively. Initialise them with empty sets.
  + Create local variable of type TreeMap<Integer,Integer> called capturedPieces and initialise it with an empty map. This will store the position a piece has moved to and the square that was jumped to get there.
  + Loop over the Direction values (using currentDir as the loop variable)
    - Use NEXT\_SQUARES[i-1] to get the next square in the current direction and store as nextSquareID.
    - If nextSquareID is not 0 switch on square
      * case BLACK\_KING or WHITE\_KING (remember kings can move in any direction)

call determineMove

* + - * case BLACK\_PIECE
        + If the move is valid for a black piece (black pieces can only go south)

call determineMove

* + - * case WHITE\_PIECE
        + If the move is valid for a white piece (white pieces can only go north)

call determineMove

* + - Construct a new MoveData object with ordinaryMoves, capturingMoves and capturedPieces and assign to a new local variable md.
    - Dependent on the colour of square and whether the md can capture add md to whiteCapturingMoves, whiteOrdinaryMoves, blackCapturingMoves or blackOrdinaryMoves
  + If whiteCapturingMoves is empty (ordinary moves are only valid if there are no capturing moves)
    - Assign whiteOrdinaryMoves to whiteMoves
  + Otherwise
    - Assign whiteCapturingMoves to whiteMoves
  + If blackCapturingMoves is empty (ordinary moves are only valid if there are no capturing moves)
    - Assign blackOrdinaryMoves to blackMoves
  + Otherwise
    - Assign blackCapturingMoves to blackMoves

### determineMove

For the specified pair of squareIDs determine if there is an ordinary or capturing move. The supplied sets and map will be updated if there is an ordinary or capturing move.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| currentSquareID | Int | The starting point for a move |
| nextSquareID | int | The next square that’ll be moved to |
| direction | Direction | The direction that the move is in |
| ordinaryMoves | Set<Integer> | Set of ordinary moves |
| capturingMoves | Set<Integer> | Set of capturing moves |
| capturedPieces | Map<Integer,Integer> | Map of new positions and the squareID captured to get there |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* If the square at nextSquareID -1 is Square.EMPTY – then you can move into it
  + Add nextSquareID to ordinaryMoves
* Otherwise if canCapture returns true for currentSquareID and direction
  + Determine the squareID after the capturing move by calling NEXT\_SQUARES[ nextSquareID -1 ].get(direction) and store in a local variable newPosn
  + Put newPosn and nextSquareID into capturedPieces
  + Add newPosn to capturingMoves

### initialBoardLayout

Sets up the board with the default layout, black pieces in squares 1 to 12 and white pieces in squares 20 to 32.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* Use Arrays.fill to set the contents of squares so that 0-12 are Square.BLACK\_PIECE, squares 21-32 are Square.WHITE\_PIECE and the rest are Square.EMPTY.
* Initialise currentPlayer to PlayerColor.BLACK.

### next

Switches to the next player.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* if currentPlayer is PlayerColor.BLACK
  + set currentPlayer to PlayerColor.WHITE
* Otherwise
  + Set currentPlayer to PlayerColor.BLACK

### processFEN

Reads a FEN tag and uses it to setup the board. It is expected that the FEN tag will only be in uppercase and has been produced from a valid board.

The fen string produced from an empty board will cause an IllegalArgumentException to be thrown as it will not contain piece sections for either player.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | IllegalArgumentException |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| fen | String | A FEN tag, in upper case only. |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* If fen is null throw an IllegalArgumentException
* Use Arrays.fill to set the contents of squares to Square.EMPTY.
* Split the fen on “:” and assign to fenElements.
* Check fenElements has length 2 or 3 – if not throw an IllegalArgumentException.
* If fenElements[0] equals “B” (first element of a fen indicates the player colour of the player whose turn it is)
  + initialise currentPlayer to PlayerColor.BLACK
* Otherwise if fenElements[0] equals “W”
  + initialise currentPlayer to new PlayerColor.WHITE
* Otherwise
  + throw an IllegalArgumentException
* Loop over fenElements from 1 using i as the loop variable (second – and optionally third – element of the fen begins with the colour of the player followed by a list of squares occupied by that colour)
  + If fenElements[i].charAt(0)==’B’
    - player = PlayerColor.BLACK
  + Otherwise if fenElements[i].charAt(0)==’W’
    - player = PlayerColor.WHITE
  + Otherwise
    - throw an IllegalArgumentException
  + Split fenElements[i].subString(1) on “,” and assign to positions
  + Loop through positions using pos as the loop variable
    - If pos has zero length continue.
    - If pos.charAt(0)==K
      * the piece is a king and use the remainder of pos as a squareID and call setSquare(). If there is a NumberFormatException throw an IllegalArgumentException detailing the problem
    - Otherwise
      * the piece is a man, use pos as a squareID and call setsquare(). If there is a NumberFormatException throw an IllegalArgumentException detailing the problem

### setSquare

Sets the contents at the specified square ID to the specified Square. This maps from 1-based square ID to 0-based array indices.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| squareID | int | The Square ID |
| contents | Square | The thing to put at squareID |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* squares[squareID-1] = contents.

## Public Fields

None

## Package Access Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Attributes | Value | Description |
| EMPTY\_LAYOUT | BoardLayout | static, final | new BoardLayout( “B:B:” ) | The board layout which is used before a game has been started |

## Protected Fields

None

## Private Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Attributes | Value | Description |
| NUMBER\_OF\_PIECES | int | static, final | 12 | The number of pieces on each player starts with. |
| NUMBER\_OF\_SQUARES | int | static, final | 32 | The number of occupiable squares on the board |
| RANKS | int[] | static, final | See below | Array of int to provide quick look up of |
| EDGE\_SQUARES | boolean[] | static, final | See below | Array to provide quick look up of edge squares. |
| NEXT\_SQUARES | NextSquare[] | static, final | See below | Array of NextSquare available for each index. |
| TURN\_LIMIT | int | static, final | 100 | The maximum number of non-taking or king moves allowed in a row. |
| blackMoves | SortedMap<Integer, MoveData> | - | - | Map describing the possible moves for the Black player |
| captured | SquareData | final | - | Describes the piece that was captured in creating this BoardLayout – will be null if no capturing move occurred. |
| squares | Square[] | final | - | Array of Square that describes the board layout. |
| turnCount | int | final | - | The number of non-taking or king moves that have happened in a row. |
| whiteMoves | SortedMap<Integer, MoveData> | - | - | Map describing the possible moves for the White player |
| currentPlayer | PlayerColor | - | - | Stores the player that can move. |

**private** **static** **final** **int**[] *RANKS* = { 1, 1, 1, 1,

2, 2, 2, 2,

3, 3, 3, 3,

4, 4, 4, 4,

5, 5, 5, 5,

6, 6, 6, 6,

7, 7, 7, 7,

8, 8, 8, 8};

**private** **final** **static** **boolean**[] *EDGE\_SQUARES* = {**true**, **true**, **true**, **true**,

**true**, **false**, **false**, **false**,

**false**, **false**, **false**, **true**,

**true**, **false**, **false**, **false**,

**false**, **false**, **false**, **true**,

**true**, **false**, **false**, **false**,

**false**, **false**, **false**, **true**,

**true**, **true**, **true**, **true**};

private static final NextSquare[] NEXT\_SQUARES = {

// Squares 1-4

new NextSquare(0, 0, 5, 6),

new NextSquare(0, 0, 6, 7),

new NextSquare(0, 0, 7, 8),

new NextSquare(0, 0, 8, 0),

// Squares 5 - 8

new NextSquare(0, 1, 0, 9),

new NextSquare(1, 2, 9, 10),

new NextSquare(2, 3, 10, 11),

new NextSquare(3, 4, 11, 12),

// Squares 9 - 12

new NextSquare(5, 6, 13, 14),

new NextSquare(6, 7, 14, 15),

new NextSquare(7, 8, 15, 16),

new NextSquare(8, 0, 16, 0),

// Squares 13 - 16

new NextSquare(0, 9, 0, 17),

new NextSquare(9, 10, 17, 18),

new NextSquare(10, 11, 18, 19),

new NextSquare(11, 12, 19, 20),

// Squares 17 - 20

new NextSquare(13, 14, 21, 22),

new NextSquare(14, 15, 22, 23),

new NextSquare(15, 16, 23, 24),

new NextSquare(16, 0, 24, 0),

// Squares 21 - 24

new NextSquare(0, 17, 0, 25),

new NextSquare(17, 18, 25, 26),

new NextSquare(18, 19, 26, 27),

new NextSquare(19, 20, 27, 28),

// Squares 25 - 28

new NextSquare(21, 22, 29, 30),

new NextSquare(22, 23, 30, 31),

new NextSquare(23, 24, 31, 32),

new NextSquare(24, 0, 32, 0),

// Squares 29 - 32

new NextSquare(0, 25, 0, 0),

new NextSquare(25, 26, 0, 0),

new NextSquare(26, 27, 0, 0),

new NextSquare(27, 28, 0, 0)};

## Public Nested Classes

None

## Nested Classes

### InvalidMoveException

|  |  |
| --- | --- |
| Attribute | Value |
| Visibility | Package |
| Modifiers | Final, Static |
| Extends | Exception |
| Implements | **-** |

#### Public Methods

None

#### Package Access Methods

None

#### Private Methods

##### InvalidMoveException

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| start | Int | Start position of the invalid move |
| end | int | End position of the invalid move |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* Invoke the super constructor with MessageFormat.format( "Move is invalid ({0}, {1})", start, end ).

#### Public Fields

None

#### Package Access Fields

None

#### Private Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Attributes | Value | Description |
| serialVersionUID | long | Static, Final | 48845639380273306L | - |

### BoardLayoutIterator

|  |  |
| --- | --- |
| Attribute | Value |
| Visibility | Private |
| Modifiers | Final, Static |
| Extends | - |
| Implements | Iterator<SquareData> |

#### Public Methods

##### hasNext

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Iterator<SquareData>.hasNext() |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| boolean | True if next() will return another SquareData, otherwise false. |

**Processing Logic**

* return !endReached.

##### next

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Iterator<SquareData>.hasNext() |
| Throws | NoSuchElementException |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| SquareData | The next piece on the board. |

**Processing Logic**

* Call hasNext(), if it returns false throw a NoSuchElementException
* Construct a new SquareData using next+1 and layout.squares[next].
* Call calcNext.
* Return the new SquareData.

##### remove

The remove functionality is not supported by this iterator and will always throw an UnsupportedOperationException

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | Iterator<SquareData>.remove |
| Throws | UnsupportedOperationException |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* Throw a new UnsupportedOperationException.

#### Package Access Methods

None

#### Private Methods

##### BoardLayoutIterator

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| layout | BoardLayout | The layout to iterate over |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* Store layout in **this**.layout
* Call calcNext()

##### calcNext

This method determines the address of the next SquareData that will be returned by next(). If the iterator has already reached the last piece also sets end.

|  |  |
| --- | --- |
| Attribute | Value |
| Modifiers | - |
| Overrides | - |
| Throws | - |

**Inputs**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| - | - | - |

**Outputs**

|  |  |
| --- | --- |
| Type | Description |
| - | - |

**Processing Logic**

* if next+1 > NUMBER\_OF\_SQUARES set endReached to true and return
* Otherwise store next in a working variable (oldValue)
* Loop through layout.squares from next+1.
  + If a non empty square is found set next to the value and break the loop.
* If next has the same value as before the loop then set endReached to true.

#### Public Fields

None

#### Package Access Fields

None

#### Private Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type | Attributes | Value | Description |
| next | int | - | -1 | Address of the next non-empty Square |
| endReached | boolean | - | false | Flag to indicate that the last piece has been found. |
| layout | BoardLayout | Final | - | The layout to iterate over |

# Resource Requirements

None

# Test Plan

## testConstructorWithNoFen

Create a board by passing in a null FEN tag. Ensure that the layout is the initial layout.

Create a board by passing in an empty string as the FEN tag. Ensure that the layout is the initial layout.

## testConstructorWithInvalidFen

Create a board by passing in the string “Invalid FEN”. Check that an IllegalArgumentException is thrown.

Create a board by passing in the string “K:B1:W3”. Check that an IllegalArgumentException is thrown.

Create a board by passing in the string “B:T1:W3”. Check that an IllegalArgumentException is thrown.

Create a board by passing in the string “K:B1:K3”. Check that an IllegalArgumentException is thrown.

## testToString

* 1. Create several different board layouts, using both constructors, and check the toString returns the expected values.

## testEquality

* 1. Create 3 identical boards.
  2. Test that equality is reflexive (boardOne equals boardOne)
  3. Test that equality is symmetric (boardOne equals boardTwo and boardTwo equals boardOne).
  4. Test that equality is transitive (boardOne equals boardTwo, boardTwo equals boardThree and boardOne equals boardThree).
  5. Test that boardOne is not equal to **null**.
  6. Create a fourth board with the same pieces as the first but with the pieces in a different order. Ensure that this board is equal to the first.
  7. Create a board that is different to the first board. Ensure that this board is not equal to the first
  8. Create a board based upon another board after a move has occurred. Ensure these boards are not equal.
  9. Create a board that matches the above board after the move has occurred. Ensure that these boards are equal.

## testHashCode

* 1. Create 3 identical boards.
  2. Test that the hashcode is reflexive (boardOne has the same hashcode as boardOne)
  3. Test that the hashcode is symmetric (boardOne has the same hashcode as boardTwo and boardTwo has the same hashcode as boardOne)
  4. Test that equality is transitive (boardOne has the same hashcode as boardTwo and boardTwo has the same hashcode as boardThree)
  5. Create a fourth board with the same pieces as the first but with the pieces in a different order. Ensure that this board has the same hashcode to the first

## testGetGameState

* 1. Create a board for each state (in progress, black win, white win) except draw. The draw state will be tested later in the test50MoveDraw.
  2. Check that getGameState returns the correct state in each case

## testInvalidMove

* 1. Check that a move is invalid if there is no piece on the starting square
  2. Check that a move is invalid if the end square is not adjacent to the start square
  3. Check that attempting to move a piece backwards is invalid
  4. Check that attempting to perform a legal move on an opponent’s piece is invalid
  5. Check that any possible ordinary move is invalid when there is a possible capturing move

## testValidMove

* 1. Check on several board layouts that valid moves do not raise an InvalidMoveException. Check the resultant board is correct.

## testCapturePiece

* 1. Check on several board layouts that valid capture moves of a piece do not raise an InvalidMoveException. Check the resultant board is correct and the correct captured piece is stored.

## testDiff

* 1. Test BoardLayout.diff returns no differences for two identical boards.
  2. Test BoardLayout.diff returns the correct differences for null first parameter.
  3. Test BoardLayout.diff returns the correct differences for null second parameter.
  4. Test BoardLayout.diff returns the correct differences for several board layouts.

## testCanSelectOrdinaryMoves

* 1. Test for a board layout where no capturing moves can be made that canSelect returns true for all squares that can be selected and false for all others.

## testCanSelectCaptureMoves

* 1. Test for a board layout with potential capturing moves that canSelect returns true for all squares that can be selected and false for all others.

## testGetRank

* 1. Check that getRank returns the correct rank for a white and black player for all squares.

## testIsEdge

* 1. Check that isEdge returns true for an edge square and false for the others.

## test50MoveDraw

* 1. Check that after 50 king moves the game is declared a draw

## testPieceResetDrawCounter

* 1. Check that if a piece moves, it takes 50 further king moves for the game to be declared a draw.

## testCaptureResetDrawCounter

* 1. Check that if a piece is captured, it takes 50 further king moves for the game to be declared a draw.

## testIterator

* 1. Create a board with the FEN tag "B:B1,K2:WK31,32"
  2. Call next() four times and ensure that the correct piece is returned and hasNext() returns the correct value.
  3. Call next() again and check that a NoSuchElementException is thrown.

## testIteratorUnsupportedOperations

* 1. Create a board with the FEN tag "B:WK32"
  2. Call remove() and check that a UnsupportedOperationException is thrown.

# Scenarios

None.

# Supporting Documentation

None.

**- End of Document -**