Table Of Content

Class PuzzleBlock

```
java.lang.Object
    |
    +--com.company.PuzzleBlock
```

All Implemented Interfaces:
com.company.BlockSortable

< Fields > < Constructors > < Methods >

public class **PuzzleBlock**
extends java.lang.Object
implements com.company.BlockSortable

Fields

extensionCount

`int` **`extensionCount`**

extensionTookPlace

`boolean` **`extensionTookPlace`**

permutation1

`java.util.List` **`permutation1`**

permutation2

`java.util.List` **`permutation2`**

permutation3

`java.util.List` **`permutation3`**

permutation4

`java.util.List` **`permutation4`**

permutation5

java.util.List **permutation5**

permutation6

java.util.List **permutation6**

rowCounter

int **rowCounter**

usedNumbers1

int[] **usedNumbers1**

usedNumbers2

int[] **usedNumbers2**

usedNumbers3

int[] **usedNumbers3**

usedNumbers4

int[] **usedNumbers4**

usedNumbers5

int[] **usedNumbers5**

usedNumbers6

int[] **usedNumbers6**

Constructors

PuzzleBlock

```
public  PuzzleBlock()
```

Methods

appearsInTheColumn

```
public boolean appearsInTheColumn(int number,
                                  int c,
                                  data.Grid grid)
```

A support method which checks if a given number appears in a column.

Parameters:

number - The number to check for.
c - The column to check at.
grid - The grid to check.

Returns:

Returns true if the number already exists somewhere in the column.

---

appearsInTheRow

```
public boolean appearsInTheRow(int number,
                               int r,
                               data.Grid grid)
```

A support method which checks if a given number appears in a row.

Parameters:

number - The number to check for.
r - The row to check at.
grid - The grid to check.

Returns:

Returns true if the number already exists somewhere in the row.

---

appearsOnceBlock

```
public boolean appearsOnceBlock(data.Grid grid)
```

This method counts the amount of times a given number appears in every block of a grid by tracking them with the variable "counter". If any of the numbers in any of the blocks appear more than once, false is returned.

Parameters:

grid - The grid to work on.

Returns:

Returns true if every number appears only once for every block.

---

appearsOnceRowCol

```
public boolean appearsOnceRowCol(data.Grid grid)
```

This method receives a grid constructed from a permutation in the method hasBlockSortColRow and checks whether every number 1-9 appears only once in every column and row. First it tracks the number of times a number has appeared in a given row with the variable "counter". Then it does the same for a given column. If in any of the numbers 1-9 appear more than once for a given column or row, return false.

Parameters:

grid - The grid to check.

Returns:

returns true if a number appears only once in every row and every column.

---

constructGrid

```
public data.Grid constructGrid(data.Grid grid,
                               java.util.List test)
```

Uses the last 9-digit permutation generated by the methods 1-6 (isBlockConflictFree etc.) and constructs the grid based on the order of the digits (which are being translated into anchor cells by the method "intToAnchor").

Parameters:

grid - The grid.
test - This list contains a permutation with the digits between 1-9

---

copyGrid

`public data.Grid` **`copyGrid`**`(data.Grid grid)`

Clones a grid by going through every row and column.

Parameters:

grid - The grid to clone.

Returns:

Returns the clone.

---

getBlockConflictFree

`public data.Grid` **`getBlockConflictFree`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers1" and selects the first available number and adds it to the List "permutation1". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether the resulting grid is h-conflict free by calling the method isBlockConflictFree and returns the grid in case it is conflict-free. If is not conflict-free it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation1". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no conflict-free grid has been found, a null-grid is returned.

Parameters:

grid - The grid to check.

Returns:

Returns a conflict-free grid if possible, otherwise a null-grid.

---

getBlockSortColRow

`public data.Grid` **`getBlockSortColRow`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers3" and selects the first available number and adds it to the List "permutation3". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether every number 1-9 appears in it only once for every column and row by calling the method "appearsOnceRowCol" and returns the grid in case it has this property. If does not, it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation3". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no grid with the property has been found, a null-grid is returned.

Parameters:

grid - The grid to work on.

Returns:

Returns "true" if there is a permutation of the grid so that every number appears once in every column and row, otherwise false.

---

getBlockSudoku

`public data.Grid` **`getBlockSudoku`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers5" and selects the first available number and adds it to the List "permutation5". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether the grid is a sudoku (i.e. every number appears only once for every unit) by calling the method "appearsOnceRowCol" and "appearsOnceBlock" and returns the grid in case it is a sudoku. If it is not, it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation5". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no sudoku has been found, a null-grid is returned.

Parameters:

grid - The grid to check.

Returns:

In case it exists, a permutation of the grid which fulfills the properties of a sudoku is returned, otherwise a null-grid.

---

hasBlockConflictFree

`public boolean` **`hasBlockConflictFree`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers2" and selects the first available number and adds it to the List "permutation2". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether the resulting grid is h-conflict free by calling the method isBlockConflictFree and returns true in case it is conflict-free. If is not conflict-free it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation2". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no conflict-free grid has been found, false is returned.

Parameters:

grid - The grid to check.

Returns:

Returns true if there is a conflict-free permutation of the grid, otherwise false.

---

hasBlockSortColRow

`public boolean` **`hasBlockSortColRow`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers4" and selects the first available number and adds it to the List "permutation4". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether every number 1-9 appears in it only once for every column and row by calling the method "appearsOnceRowCol" and returns the grid in case it has this property. If does not, it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation4". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no grid with the property has been found, a null-grid is returned.

Parameters:

grid - The grid to work on.

Returns:

Returns true if there is a grid permutation where every number appears once for every column and row, otherwise false.

---

hasBlockSudoku

`public boolean` **`hasBlockSudoku`**`(data.Grid grid)`

This method recursively creates all possible permutations of the numbers between 0-8. It always memorizes which of the numbers 0-8 have been used so far (as they may appear only once) in the variable "usedNumbers6" and selects the first available number and adds it to the List "permutation6". After finishing one recursion, it checks if the length of the list is 9. In case it is, it calls the method constructGrid which uses the 9 values (which stand for the 9 anchor cells) to construct a full grid. Then it checks whether the grid is a sudoku (i.e. every number appears only once for every unit) by calling the method "appearsOnceRowCol" and "appearsOnceBlock" and returns "true" in case it is a sudoku. If it is not, it returns the number p back to the available numbers and takes the next available number from the list to make up a different permutation in "permutation6". So the development of the permutations is as follows: 012345678, 012345687, 012345768, 012345786, 012345867, 012345876 etc. In case after going through all possible permutations still no sudoku has been found, "false" is returned.

Parameters:

grid - The grid to work on.

Returns:

Returns true if there is a permutation of the grid which fulfills the properties of a sudoku.

---

intToAnchor

`public int[]` **`intToAnchor`**`(int i)`

This method interprets a number between 0-8 as coordinates of one of the 9 anchor cells. E.g. 3 (fourth number) represents the anchor cell with coordinates 4,1 so a two-dimensional array consisting {4,1} is returned.

Parameters:

i - A digit from 0-8 representing a block in a 9x9 grid.

Returns:

Returns a two-dimensional array which translates 0 into 1,1; 1 into 1,4 and so on.

---

isBlockConflictFree

`public boolean` **`isBlockConflictFree`**`(data.Grid grid)`

Compares the 3th with the 4th and the 6th with the 7th cell of every row in the grid. If any of them match, the grid is not h-conflict free. All the other neighboring cells (e.g. 1st and 2nd column) are not checked because we assume that the blocks themselves are h-conflict free.

Parameters:

grid - The grid we are working on.

Returns:

Returns true if the given grid is h-conflict free.

---

nullGrid

```
public data.Grid nullGrid()
```

Constructs a Grid filled with zeros.

Returns:

Returns a 9x9 grid filled with zeros.

---

putNumberColRow

```
public int putNumberColRow(data.Grid grid,
                           int number)
```

The method goes through the grid row-wise using the variable "rowCounter". "Number" must not already appear anywhere in the current row. If it does, go to the next row. Then, by going through every cell of the current row, search for the first available cell which is empty and for which the property applies that it must not appear in the column. If all these conditions are met, "number" is being put and we proceed to the next row and a recursive call is being made. Once rowCounter reaches 10, it is time to increase "extensionCount" by one, but only in case any number has been put (which is being tracked by "extensionTookPlace), i.e. only in case any changes to the grid have been made. Because of its recursive nature, the method eventually goes through all possibilities in the same pattern as the first 6 methods do.

Parameters:

grid - The grid to work on.
number - The number to use in the extension (Erweiterung).