```
package hw3;

import static api.Direction.*;
import static api.Orientation.*;

import java.util.ArrayList;

import api.Cell;
import api.Direction;
import api.Move;
import api.Orientation;

/**
 * Represents a board in the Block Slider game. A board contains a 2D grid of
 * cells and a list of blocks that slide over the cells.
 * @author Maxwell Skinner
 */
public class Board {
	/**
	 * Tells the cell that is grabbed
	 */
	private Cell cellGrabbed;
	/**
	 * Tells the block the player has grabbed.
	 */
	private Block blockGrabbed;
	/**
	 * The current number of moves the player has performed.
	 */
	private int moveCount;
	/**
	 * 2D array of cells, the indexes signify (row, column) with (0, 0)
representing
	 * the upper-left corner of the board.
	 */
	private Cell[][] grid;

	/**
	 * A list of blocks that are positioned on the board.
	 */
	private ArrayList<Block> blocks;

	/**
	 * A list of moves that have been made in order to get to the current
position
	 * of blocks on the board.
	 */
	private ArrayList<Move> moveHistory;

	/**
	 * Constructs a new board from a given 2D array of cells and list of blocks.
The
	 * cells of the grid should be updated to indicate which cells have blocks
	 * placed over them (i.e., setBlock() method of Cell). The move history
should
	 * be initialized as empty.
	 *
	 * @param grid   a 2D array of cells which is expected to be a rectangular
shape
```

```
	 * @param blocks list of blocks already containing row-column position which
	 *               should be placed on the board
	 */
	public Board(Cell[][] grid, ArrayList<Block> blocks) {
		this.grid = new Cell[grid.length][grid[0].length];
		cellGrabbed = null;
		blockGrabbed = null;
		moveCount = 0;
		moveHistory = new ArrayList<Move>();
		for(int i = 0; i< grid.length; i += 1) {
			for(int j = 0; j < grid[i].length; j += 1) {
				this.grid[i][j] = grid[i][j];
			}
		}
		// Runs through the grid to find blocks and adds them to the blocks
array for this class.
			this.blocks = new ArrayList<>(blocks);
			for(int i = 0; i < blocks.size(); i += 1) {
			Block holder = blocks.get(i);
			int blockRow= holder.getFirstRow();
			int blockCol = holder.getFirstCol();
			Orientation holderOrient = holder.getOrientation();
			grid[blockRow][blockCol].setBlock(holder);
			if(holderOrient == Orientation.VERTICAL) {
				for(int j = 1; j < holder.getLength(); j += 1) {
					grid[blockRow + j][blockCol].setBlock(holder);
				}
			}
			else if(holder.getOrientation()== Orientation.HORIZONTAL &&
holder.getLength()>1){
				for(int j = 1; j < holder.getLength(); j += 1) {
					grid[blockRow][blockCol + j].setBlock(holder);
				}
			}
		}
	}

	/**
	 * Constructs a new board from a given 2D array of String descriptions.
	 * <p>
	 * DO NOT MODIFY THIS CONSTRUCTOR
	 *
	 * @param desc 2D array of descriptions
	 */
	public Board(String[][] desc) {
		this(GridUtil.createGrid(desc), GridUtil.findBlocks(desc));
	}

	/**
	 * Models the user grabbing a block over the given row and column. The
purpose
	 * of grabbing a block is for the user to be able to drag the block to a new
	 * position, which is performed by calling moveGrabbedBlock(). This method
	 * records two things: the block that has been grabbed and the cell at which
it
	 * was grabbed.
	 *
	 * @param row row to grab the block from
	 * @param col column to grab the block from
```

```
	 */
	public void grabBlockAtCell(int row, int col) {
		blockGrabbed = grid[row][col].getBlock();
		cellGrabbed = getCell(row, col);
	}

	/**
	 * Set the currently grabbed block to null.
	 */
	public void releaseBlock() {
		blockGrabbed = null;
		cellGrabbed = null;
	}

	/**
	 * Returns the currently grabbed block.
	 *
	 * @return the current block
	 */
	public Block getGrabbedBlock() {
		return blockGrabbed;
	}

	/**
	 * Returns the currently grabbed cell.
	 *
	 * @return the current cell
	 */
	public Cell getGrabbedCell() {
		return cellGrabbed;
	}

	/**
	 * Returns true if the cell at the given row and column is available for a
block
	 * to be placed over it. Blocks can only be placed over floors and exits. A
	 * block cannot be placed over a cell that is occupied by another block.
	 *
	 * @param row row location of the cell
	 * @param col column location of the cell
	 * @return true if the cell is available for a block, otherwise false
	 */
	public boolean canPlaceBlock(int row, int col) {
		if((grid[row][col].isFloor()||grid[row][col].isExit())&&(grid[row]
[col].hasBlock()==false)) {
		return true;
	}
		else {
			 return false;
		}
	}

	/**
	 * Returns the number of moves made so far in the game.
	 *
	 * @return the number of moves
	 */
	public int getMoveCount() {
		return moveCount;
```

```
}

/**
 * Returns the number of rows of the board.
 *
 * @return number of rows
 */
public int getRowSize() {
      return grid.length;
}

/**
 * Returns the number of columns of the board.
 *
 * @return number of columns
 */
public int getColSize() {
      return grid[0].length;
}

/**
 * Returns the cell located at a given row and column.
 *
 * @param row the given row
 * @param col the given column
 * @return the cell at the specified location
 */
public Cell getCell(int row, int col) {
      // TODO
      return grid[row][col];
}

/**
 * Returns a list of all blocks on the board.
 *
 * @return a list of all blocks
 */
public ArrayList<Block> getBlocks() {
      return blocks;
}

/**
 * Returns true if the player has completed the puzzle by positioning a block
 * over an exit, false otherwise.
 *
 * @return true if the game is over
 */
public boolean isGameOver() {
      for(int i = 0; i<grid.length; i += 1) {
            for(int j = 0; j <grid[i].length; j += 1) {
                  if(grid[i][j].isExit() && grid[i][j].hasBlock()) {
                        return true;
                  }
            }
      }
      return false;
}

/**
```

```
	 * Moves the currently grabbed block by one cell in the given direction. A
	 * horizontal block is only allowed to move right and left and a vertical
block
	 * is only allowed to move up and down. A block can only move over a cell
that
	 * is a floor or exit and is not already occupied by another block. The
method
	 * does nothing under any of the following conditions:
	 * <ul>
	 * <li>The game is over.</li>
	 * <li>No block is currently grabbed by the user.</li>
	 * <li>A block is currently grabbed by the user, but the block is not allowed
to
	 * move in the given direction.</li>
	 * </ul>
	 * If none of the above conditions are meet, the method does the following:
	 * <ul>
	 * <li>Moves the block object by calling its move method.</li>
	 * <li>Sets the block for the grid cell that the block is being moved
into.</li>
	 * <li>For the grid cell that the block is being moved out of, sets the block
to
	 * null.</li>
	 * <li>Moves the currently grabbed cell by one cell in the same moved
direction.
	 * The purpose of this is to make the currently grabbed cell move with the
block
	 * as it is being dragged by the user.</li>
	 * <li>Adds the move to the end of the moveHistory list.</li>
	 * <li>Increment the count of total moves made in the game.</li>
	 * </ul>
	 *
	 * @param dir the direction to move
	 */
	//If you move the blocks in the UI using the farthest left(Horizontal Blocks)
	//or farthest up (Vertical Blocks) the game will always work.
	public void moveGrabbedBlock(Direction dir) {
			if((isGameOver()==false)&&(getGrabbedBlock() != null)) {
				Block movingBlock = getGrabbedBlock();
				int blockRow = movingBlock.getFirstRow();
				int blockColumn = movingBlock.getFirstCol();
				int blockLength = movingBlock.getLength();
				Orientation grabbedOrientation =
movingBlock.getOrientation();
				//Based on the direction, the code will call a helper
method associated with
				//the corresponding direction.
				if((dir == RIGHT) && (grabbedOrientation == HORIZONTAL)) {
					right(movingBlock, blockLength, blockRow,
blockColumn);
				}
				if((dir== LEFT) && (grabbedOrientation == HORIZONTAL)) {
					left(movingBlock, blockLength, blockRow,
blockColumn);
				}
				if((dir == UP) && (grabbedOrientation == VERTICAL)) {
					up(movingBlock, blockLength, blockRow, blockColumn);
				}
				if((dir == DOWN) && (grabbedOrientation == VERTICAL)) {
```

```
                              down(movingBlock, blockLength, blockRow,
blockColumn);
                        }
                  }
            }
      /**
       * Helper method used to adjust the grabbed block right.
       * @param movingBlock The block that is going to move.
       * @param blockLength  The length of the moving block.
       * @param blockRow The first row of the moving block.
       * @param blockColumn  The first column of the moving block.
       */
      private void right(Block movingBlock, int blockLength, int blockRow, int
blockColumn) {
            int endBlock = blockColumn + blockLength-1;
            if(canPlaceBlock(blockRow, endBlock+1)) {
                  movingBlock.move(RIGHT);
                  grid[blockRow][blockColumn].clearBlock();
                  grid[blockRow][endBlock+1].setBlock(movingBlock);

                  Block newBlock = new Block(movingBlock.getFirstRow(),
movingBlock.getFirstCol(), movingBlock.getLength(), movingBlock.getOrientation());
                  moveHistory.add(new Move(newBlock, RIGHT));
                  grabBlockAtCell(blockRow, blockColumn +1);
                  moveCount += 1;
            }
      }
      /**
       * Helper method used to adjust the grabbed block left.
       * @param movingBlock The block that is going to move.
       * @param blockLength  The length of the moving block.
       * @param blockRow The first row of the moving block.
       * @param blockColumn  The first column of the moving block.
       */
      private void left(Block movingBlock, int blockLength, int blockRow, int
blockColumn) {
            int endBlock = blockColumn + blockLength-1;
            if(canPlaceBlock(blockRow, blockColumn-1)) {
                  movingBlock.move(LEFT);
                  grid[blockRow][endBlock].clearBlock();
                  grid[blockRow][blockColumn-1].setBlock(movingBlock);

                  Block newBlock = new Block(movingBlock.getFirstRow(),
movingBlock.getFirstCol(), movingBlock.getLength(), movingBlock.getOrientation());
                  moveHistory.add(new Move(newBlock, LEFT));
                  grabBlockAtCell(blockRow, blockColumn - 1);
                  moveCount += 1;
            }
      }
      /**
       * Helper method used to adjust the grabbed block up.
       * @param movingBlock The block that is going to move.
       * @param blockLength  The length of the moving block.
       * @param blockRow The first row of the moving block.
       * @param blockColumn  The first column of the moving block.
       */
      private void up(Block movingBlock, int blockLength, int blockRow, int
blockColumn) {
            int endBlock = blockRow + blockLength-1;
```

```
            if(canPlaceBlock(blockRow-1, blockColumn)) {
                  movingBlock.move(UP);
                  grid[endBlock][blockColumn].clearBlock();
                  grid[blockRow-1][blockColumn].setBlock(movingBlock);

                  Block newBlock = new Block(movingBlock.getFirstRow(),
movingBlock.getFirstCol(), movingBlock.getLength(), movingBlock.getOrientation());
                  moveHistory.add(new Move(newBlock, UP));
                  grabBlockAtCell(blockRow-1, blockColumn);
                  moveCount += 1;
            }
      }
      /**
       * Helper method used to adjust the grabbed block down.
       * @param movingBlock The block that is going to move.
       * @param blockLength  The length of the moving block.
       * @param blockRow The first row of the moving block.
       * @param blockColumn  The first column of the moving block.
       */
      private void down(Block movingBlock, int blockLength, int blockRow, int
blockColumn) {
            int endBlock = blockRow + blockLength-1;
            if(canPlaceBlock(endBlock+1, blockColumn)) {
                  movingBlock.move(DOWN);
                  grid[blockRow][blockColumn].clearBlock();
                  grid[endBlock+1][blockColumn].setBlock(movingBlock);

                  Block newBlock = new Block(movingBlock.getFirstRow(),
movingBlock.getFirstCol(), movingBlock.getLength(), movingBlock.getOrientation());
                  moveHistory.add(new Move(newBlock, DOWN));
                  grabBlockAtCell(blockRow+1, blockColumn);
                  moveCount += 1;
            }
      }

      /**
       * Resets the state of the game back to the start, which includes the move
       * count, the move history, and whether the game is over. The method calls
the
       * reset method of each block object. It also updates each grid cells by
calling
       * their setBlock method to either set a block if one is located over the
cell
       * or set null if no block is located over the cell.
       */
      public void reset() {
            for(int i = moveHistory.size() - 1; i >=0; i -= 1) {
                  Move moveUndo = moveHistory.get(i);
                  Block oldBlock = moveUndo.getBlock();
                  Direction oldDirection = moveUndo.getDirection();
                  int blockRow = oldBlock.getFirstRow();
                  int blockColumn = oldBlock.getFirstCol();
                  grabBlockAtCell(blockRow, blockColumn);
                  // Resets the game by doing the player's moves in reverse then
clearing the move history.
                  Block grabbedBlock = getGrabbedBlock();
                  int blockLength2 = grabbedBlock.getLength();
                  int blockRow2 = grabbedBlock.getFirstRow();
                  int blockColumn2 = grabbedBlock.getFirstCol();
```

```
                  if(oldDirection== RIGHT) {
                        left(grabbedBlock, blockLength2, blockRow2, blockColumn2);
                  }
                  if(oldDirection== LEFT) {
                        right(grabbedBlock, blockLength2, blockRow2, blockColumn2);
                  }
                  if(oldDirection== UP){
                        down(grabbedBlock, blockLength2, blockRow2, blockColumn2);
                  }
                  if(oldDirection== DOWN) {
                        up(grabbedBlock, blockLength2, blockRow2, blockColumn2);
                  }
            }
            moveHistory.clear();
            blockGrabbed = null;
            cellGrabbed = null;
            moveCount = 0;
      }

      /**
       * Returns a list of all legal moves that can be made by any block on the
       * current board. If the game is over there are no legal moves.
       *
       * @return a list of legal moves
       */
      public ArrayList<Move> getAllPossibleMoves() {
            ArrayList<Move> possibleMoves = new ArrayList<>();
            if(isGameOver()==true) {
                  possibleMoves.clear();
            return possibleMoves;
            }
            else {
                  for(int i = 0; i<blocks.size(); i += 1) {
                        Orientation check = blocks.get(i).getOrientation();
                        if(check == HORIZONTAL) {
                              if(canPlaceBlock(blocks.get(i).getFirstRow(),
blocks.get(i).getFirstCol()-1)) {
                                    possibleMoves.add(new Move(blocks.get(i),
LEFT));
                              }
                              if(canPlaceBlock(blocks.get(i).getFirstRow(),
blocks.get(i).getFirstCol()+blocks.get(i).getLength())) {
                                    possibleMoves.add(new Move(blocks.get(i),
RIGHT));
                              }
                        }
                        else if(check == VERTICAL){
                              if(canPlaceBlock(blocks.get(i).getFirstRow()-1,
blocks.get(i).getFirstCol())) {
                                    possibleMoves.add(new Move(blocks.get(i), UP));
                              }
                              if(canPlaceBlock(blocks.get(i).getFirstRow()
+blocks.get(i).getLength(), blocks.get(i).getFirstCol())) {
                                    possibleMoves.add(new Move(blocks.get(i),
DOWN));
                              }
                        }
                  }
```

```
            }
            return possibleMoves;
      }

      /**
       * Gets the list of all moves performed to get to the current position on the
       * board.
       *
       * @return a list of moves performed to get to the current position
       */
      public ArrayList<Move> getMoveHistory() {
            return moveHistory;
      }

      /**
       * EXTRA CREDIT 5 POINTS
       * <p>
       * This method is only used by the Solver.
       * <p>
       * Undo the previous move. The method gets the last move on the moveHistory
list
       * and performs the opposite actions of that move, which are the following:
       * <ul>
       * <li>grabs the moved block and calls moveGrabbedBlock passing the opposite
       * direction</li>
       * <li>decreases the total move count by two to undo the effect of calling
       * moveGrabbedBlock twice</li>
       * <li>if required, sets is game over to false</li>
       * <li>removes the move from the moveHistory list</li>
       * </ul>
       * If the moveHistory list is empty this method does nothing.
       */
      public void undoMove() {
            moveCount -= 1;
      }

      @Override
      public String toString() {
            StringBuffer buff = new StringBuffer();
            boolean first = true;
            for (Cell row[] : grid) {
                  if (!first) {
                        buff.append("\n");
                  } else {
                        first = false;
                  }
                  for (Cell cell : row) {
                        buff.append(cell.toString());
                        buff.append(" ");
                  }
            }
            return buff.toString();
      }
}
```