Com S 227
Spring 2022
Assignment 3
300 points

Due Date: Wednesday, April 20, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm April 19)

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details. You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

Contents

Introduction

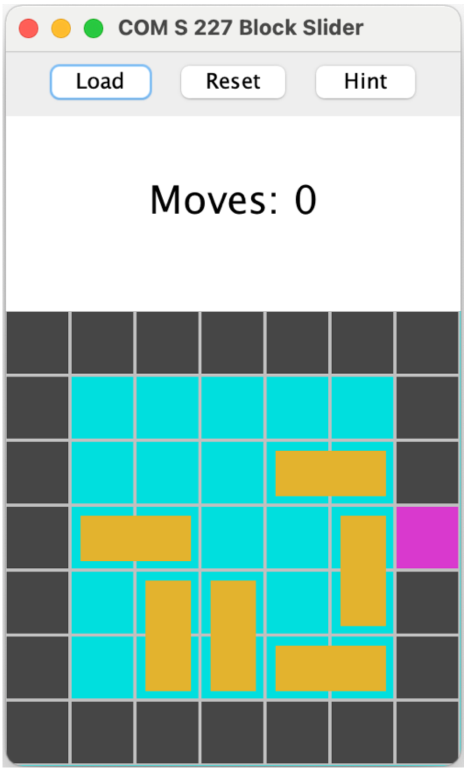
The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and, most importantly, to get some experience putting together a working application involving several interacting Java classes.

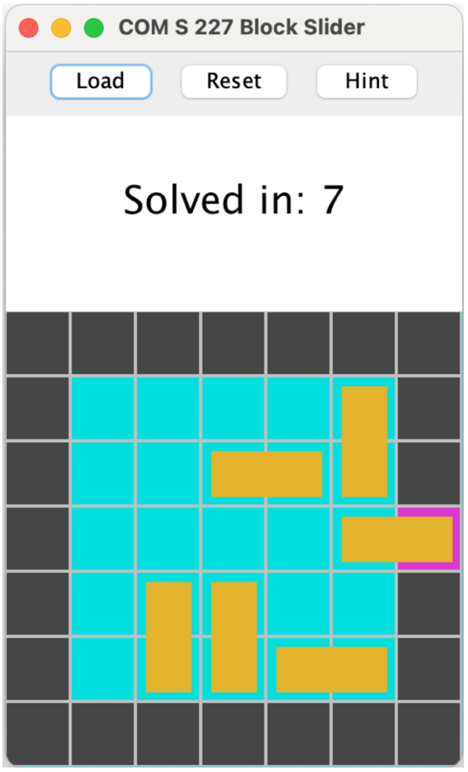
There are three classes for you to implement: **`Block, Board, and GridUtil`**. There is also an extra credit class **`Solver`** that is covered in a separate document. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

The three classes can be used, along with some other components, to create an implementation of our version of a sliding block puzzle game.

The game is played on a 2-dimensional grid of cells. Each cell may be a floor, a wall or exit. Cells that are floors or exits can be occupied by a block. Blocks are 1 cell wide by 2 or more cells long and are oriented either horizontally or vertically. A horizontal block can only move left and right and a vertical block can only move up and down. Only one block can occupy a cell at a time.

The goal is to move one of the blocks to the exit. Naturally, other blocks may be in the path and need to be moved. A block can be moved by grabbing (pressing the mouse button) on any of its segments and dragging. The block is move one cell at a time as it is being dragged.





The three classes you implement will provide the "backend" or core logic for the game. In the interest of having some fun with it, we will provide you with code for a GUI (graphical user interface), based on the Java Swing libraries.

The sample code includes a documented skeleton for the classes you are to create in the package **`hw3`**. The additional code is in the packages **`ui`** and **`api`**. The **`ui`** package is the code for the GUI, described in more detail in a later section. The **`api`** package contains some relatively boring types for representing data in the game, along with a utility class with methods for creating the game grid and printing out the game in text format. There are a few examples of test cases for you to start with located in the default package, along with a simple text-based user interface.

You should not modify the code in the **`api`** package.

Specification

The complete specification for this assignment includes

- this pdf,
- a pdf describing the extra credit solver,
- the online Javadoc, and
- any "official" clarifications posted on Canvas and/or Piazza

The enumerated types Orientation, Direction, and CellType

Several constants are used to describe the state and actions of the games. To describe a blocks orientation of the board, we use two constants **`Orientation.VERTICAL`** and **`Orientation.HORIZONTAL`**. In addition, the motion of a block is described using four constants **`Direction.LEFT`**, **`Direction.RIGHT`**, **`Direction.UP`**, and **`Direction.DOWN`**. Finally, types of cells are described as **`CellType.FLOOR`**, **`CellType.WALL`**, and **`CellType.EXIT`**.

All of the above constants are defined as an "enumerated" type or **`enum`**. You can basically use them as though they had been defined as **`public static final`** constants. You can't construct instances, you just use the four names as literal values. Use **`==`** to when checking whether two values of this type are the same, e.g. if **`dir`** is a variable of type **`Direction`**, you can write things like

```
if (dir == Direction.UP)
{
    // do cool stuff
}
```

An **`enum`** type can also be used as the switch expression in a **`switch`** statement.

Tip: add the lines
```
import static api.Orientation.*;
import static api.Direction.*;
import static api.CellType.*;
```

to the top of your file. Then you can refer to the constants without having to type, for example, "`Direction.`" in front of them.

Overview of the Block class

A **`Block`** object is described by its location on the board (first row and column), its length and its orientation (vertical or horizontal). The first row and column indicate the location of the upper-left most corner of the block. In other words, vertical blocks extend downward and horizontal blocks extend to the right of the first row and column.

The **`move`** method can be used to move a horizontal block one cell right or left and a vertical block one cell up or down. Here moving simply means the first row and column of the block are updated.

A **`reset`** method simply resets the first row and column to their original starting point when the object was created.

Overview of the Board class

A **`Board`** object maintains a 2D array of cells, a list of blocks, and other state related to the game. The user can “grab” a block on the board with the method **`grabBlockAtCell`**, which is called by the GUI when a user presses the mouse button down on any segment of a block. The grabbed block can be dragged one cell at a time either up, down, right, or left by calling **`moveGrabbedBlock`**. When moving a block, the state of the cells and the block must both be updated to be consistent. This is because the cells know what block is located over them and the blocks know their row and column location on the board. An important piece of information is the cell over which the block is being grabbed. The method **`getGrabbedCell`** is required by the GUI to determine when the block is being dragged one more cell. This means that the currently grabbed cell also moves by one in the same direction each time the grabbed block is moved. Each move is also added as a new **`Move`** object to the end of a list **`moveHistory`**. The user can end grabbing the block by calling **`releaseBlock`**.

The board can be reset to the starting point when it was created by calling **`reset`**. The blocks, cells, move count, move history, and is game over status should all be returned to their original starting values.

The method get **`getAllPossibleMoves`** finds all legal moves that can be made from the current position and returns it as a list. This method is used by the hint feature to suggest a random legal move that can be made. It will also be useful when implementing the game solver described below.

One final method, which is useful when implementing the solver, is **`undo`**. As its name implies it undoes the most recent move, setting the blocks position and move count back to their previous values.

Overview of the GridUtil class

The **`GridUtil`** class contains two static methods used for converting a string description of a board into a 2D array of cells and a list of blocks. The following is an example of a board description.

```
* * * * * * * *
* [ ] ^ ^ . . *
* . . v v [ ] *
* [ # # ] ^ . *
* ^ [ ] . v ^ e
* # ^ . [ ] # *
* # v . [ ] v *
* v [ # # # ] *
* * * * * * * *
```

The meaning of the symbols are:

- **`*`** represents a wall
- **`e`** represents an exit
- `.` represents a floor
- Horizontal blocks are indicated by **`[ # # ]`** with zero or more **`#`**.
- Vertical blocks are indicated by:

```
^
#
#
v
```

  with zero or more **`#`**.

Overview of the Solver class (extra credit)

A **`Solver`** object uses recursion to find all solutions to a board setup. Completing the class is extra credit. Suggestions on how to implement the solve method are provided in a separate document.

The GUI

There is also a graphical UI in the **`ui`** package. The GUI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course. You are not expected to be able to read and understand it. However, you might be interested in exploring how it works at some point. In particular it is sometimes helpful to look at how the UI is calling the methods of the classes you are writing.

The controls are simple: You can press down on the mouse to grab a any segment of a block and drag to move the block.

The main method is in `ui.GameMain`. You can try running it, and you'll see the initial window, but until you start implementing the required classes you'll just get errors. All that the main class does is to initialize the components and start up the UI machinery. The class `GamePanel` contains most of the UI code and defines the "main" panel, and there is also a much simpler class `ScorePanel` that contains the display of the score.

You can edit `GameMain` to use a different initial board descriptor. Alternatively, once you have `GridUtil` fully implemented, you can use the "Choose from file" button in the game to select one of the games from a file of descriptors.

The graphical board is a representation of the 2D array of api.Cell object and the list of blocks in Board.

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The code for the UI itself is more complex than the code you are implementing, and it is not guaranteed to be free of bugs***. In particular, when we grade your work we are NOT going to run the UI, we are going to test that each method works according to its specification.***

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared `private`, and if you want to add any additional "helper" methods that are not specified, they must be declared `private` as well.

Importing the sample code

The sample code includes a complete skeleton of the four classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box.

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the `hw3`, `ui`, and `api` folders from Explorer/Finder into the `src` folder in Eclipse.
6. Copy the remaining top-level files (SimpleTest.java, etc) into the `src` folder. You will also want `games.txt`.

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification, so this getting started section will not be quite as detailed as for the previous assignments.

You can find the example test cases below in the default package of the sample code. *Don't try to copy/paste from the pdf.*

Please remember that these are just examples to get you started and you will need to write many, many more tests to be sure your code is working correctly. (You can expect that the functional tests we run when grading your work will have something like 100 test cases, for example.)

Don't rely on the GUI for testing your code. Doing so is difficult and frustrating because the GUI itself is so complex. Rely on simple, focused test cases that you can easily run in the debugger.

0. **Board** is largely dependent on **Block**, while **Block** has no dependencies on the rest of the **hw3** package. Also, **Block** is mostly simple accessors that can be finished quickly, so it is a good starting point. The only method you might want to test at this point is **move**. The following code from **SimpleTest.java** shows how you can test **Block**.

```
Block block = new Block(2, 1, 2, HORIZONTAL);
System.out.println("Block is " + block);
block.move(DOWN); // horizontal blocks not allowed to move down
System.out.println("After move DOWN, block is " + block);
System.out.println("Expected block at (row=2, col=1).");
block.move(RIGHT);
System.out.println("After move RIGHT, block is " + block);
System.out.println("Expected block at (row=2, col=2).");
```

1. **GridUtil** also has no dependencies on the **hw3** package, so you can work on it at any time. Setting up tests may be easier with a working **GridUtil**, so now is a good time to complete the implementation. The following code from **SimpleTest.java** shows how you can test **GridUtil**.

Before main:
```
public static final String[][] testDescription1 =
            { { "*", "*", "*", "*", "*" },
              { "*", ".", ".", ".", "*" },
              { "*", "[", "]", ".", "e" },
              { "*", ".", ".", ".", "*" },
              { "*", "*", "*", "*", "*" } };
```

In main:
```
Cell[][] cells = GridUtil.createGrid(testDescription1);
System.out.println("Using testDescription1, cell (2, 4) is an exit is "
            + cells[2][4].isExit() + ", expected is true.");

ArrayList<Block> blocks = GridUtil.findBlocks(testDescription1);
System.out.println("Using testDescription1, number of blocks is "
            + blocks.size() + ", expected is 1.");
System.out.println("Using testDescription1, first block is length "
            + blocks.get(0).getLength() + ", expected is 2.");
```

2. Now you are ready to begin on the largest class, **Board**. The constructor for Board is a natural starting point. The following code from **SimpleTest.java** shows how you can test the **Board** constructor.

Before main:
```
private static ArrayList<Block> makeTest1Blocks() {
      ArrayList<Block> blocks = new ArrayList<Block>();
      blocks.add(new Block(2, 1, 2, HORIZONTAL));
      return blocks;
}
```

In main:
```
System.out.println("Making board with testGrid1.");
Board board = new Board(testGrid1, makeTest1Blocks());
System.out.println(board.toString());
```

The expected output is:

```
Making board with testGrid1.
* * * * *
* . . . *
* [ ] . e
* . . . *
* * * * *
```

3. Moving blocks on the board requires the ability to grab a block. At this point it would make sense to complete **`grabBlockAtCell`**, **`releaseBlock`**, **`getGrabbedBlock`**, **`getGrabbedCell`**. These methods deal with tracking which block is currently "grabbed" by the user and which cell is currently bellow the grabbed segment of the block. The following code from **`SimpleTest.java`** shows how you can test **`grabBlockAtCell`**.

```
board.grabBlockAtCell(2, 1);
System.out.println("Grabbed block " + board.getGrabbedBlock());
System.out.println("Location of grab is at ("
      + board.getGrabbedCell().getRow()
      + ", " + board.getGrabbedCell().getCol()
      + "), expected (2, 1).");
```

4. The last method to implement before working on **`moveGrabbedBlock`** is **`canPlaceBlock`**. This method is a useful helper method for determining if it is possible for a block to be placed in a particular location on the board.

5. Now you are ready to implement **`moveGrabbedBlock`**. This method has a lot of things it needs to do as described in the specification. For this reason, it may help to break the implementation up into smaller helper methods. The following code from **`SimpleTest.java`** shows how you can test **`moveGrabbedBlock`**.

```
board.moveGrabbedBlock(RIGHT);
System.out.println("After moving block right one time game over is "
      + board.isGameOver() + ", expected is false.");
System.out.println(board.toString());
System.out.println();

board.moveGrabbedBlock(RIGHT);
System.out.println("After moving block right two times game over is "
      + board.isGameOver() + ", expected is true.");
System.out.println(board.toString());
```

The expected output is:

```
Using testGrid1, after moving block right one time game over is false,
expected is false.
* * * * *
* . . . *
* . [ ] e
* . . . *
* * * * *
```

```
Using testGrid1, after moving block right two times game over is true,
expected is true.
* * * * *
* . . . *
* . . [ e
* . . . *
* * * * *
```

By now there should be enough functionality implemented that the GUI is mostly working. You can start it from **`ui/GameMain.java`**. Several example games are provided with the project skeleton code in **`games.txt`**. Keep in mind the warning above that a GUI can't replace test cases when debugging and testing code, but it can be helpful to visualize the game.

6. The most significant remaining methods are **`reset`** and **`getAllPossibleMoves`**. These can be implemented in either order.

```
board.reset();
System.out.println("After reset:");
System.out.println(board.toString());
System.out.println();

ArrayList<Move> moves = board.getAllPossibleMoves();
System.out.println("All possible moves: " +
Arrays.toString(moves.toArray()));
```

The expected output is:

```
After reset:
* * * * *
* . . . *
* [ ] . e
* . . . *
* * * * *

All possible moves: [(2, 1) one cell RIGHT]
```

7. Now finish the remaining methods in **`Board`**.

8. If you are ready, you can now move on to the extra credit methods **`undoMove`** in **`Board`** and **`solve`** in **`Solver`**. Information about these methods is provided in a separate document.

The SpecChecker

You can find the SpecChecker on Canvas. Import and run the SpecChecker just as you practiced in Lab 1. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the *console* output. There are many test cases so there may be an overwhelming number of error messages. ***Always start***

reading the errors at the top and make incremental corrections in the code to fix them. When you are happy with your results, click "Yes" at the dialog to create the zip file. See the document "SpecChecker HOWTO", if you are not sure what to do.

Style and documentation

Roughly 10% of the points will be for documentation and code style. The general guidelines are the same as in homework 2. However, since the skeleton code has the public methods fully documented, there is not quite as much to do. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the classes you write (in this case the classes in the hw3 package).
- You must javadoc each instance variable and helper method that you add. Anything you add must be **`private`**.
- Since the code includes some potentially tricky loops to write, **you ARE expected to add internal (//-style) comments, where appropriate**, to explain what you are doing inside the longer methods. A good rule of thumb is: if you had to think for a few minutes to figure out how to do something, you should probably include a comment explaining how it works. Internal comments always *precede* the code they describe and are indented to the same level.
- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`hw3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any announcements from the that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw3.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains four files, **`Block.java`**, **`Board.java`**, **`GridUtil.java`**, and **`Solver.java`**. Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 3 submission link and **VERIFY** that your submission was successful.

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the three required files. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.