Homework 4: BlockAddiction!!

COMS 2270 Fall 2024

Contents

General Information

- 300 Points
- Due Date: Friday December 13th 2024, 11:59 pm (midnight).
- As usual there will be a 5% bonus for submitting a day early, by Thursday December 12th 2024 at 11:59pm.
- NO LATE SUBMISSIONS - EVERYTHING MUST BE IN FRIDAY NIGHT
- This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus for details. If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions. Post all your related questions under the "hw4" topic on Piazza. Please start the assignment as soon as possible and get your questions answered right away!

The purpose of this assignment is to give you some experience working with inheritance and abstract classes in a realistic setting. You'll also get some more practice with 2D arrays and lists.

- Create implementations of six concrete subtypes of the Piece interface:
  - LShapedPiece
  - TeePiece
  - QuotesPiece
  - RotatingSPiece
  - CirclingPiece
  - SnakingPiece
- Implement the abstract class **AbstractPiece** containing the common code for the concrete Piece classes above, and any other abstract classes that express general code common to two or more (but not all) of the pieces above.

All your code should be in the package **hw4**.

TIPS FROM THE EXPERTS

HOW TO WASTE A LOT OF TIME ON THIS ASSIGNMENT

- Start the assignment the night it's due. That way, if you have questions, the TAs will be too busy to help you and you can spend the time tearing your hair out over some trivial detail.
- Don't bother reading the rest of this document, or even the specification, especially not the "Getting started" section. Documentation is for losers. Try to write lots of code before you figure out what it's supposed to do.
- Don't test your code. It's such fun to remain in suspense until it's graded!

OVERVIEW

In this project you will complete the implementation of a Tetris-style or "falling blocks" game. This game, which we will call BlockAddiction, is a mix of "Tetris" with a game like "Bejeweled". If you are not familiar with such games, examples are easy to find on the internet.

FIRST – PLAY THE GAME BY RUNNING THE JAR FILE PROVIDED TO YOU. Type on the command line: `java -jar tetris.jar`. Note that the pieces provided in the `tetris.jar` file are mostly different from the ones you will implement.

The basic idea is as follows. The game is played on a 2D grid. Each position in this grid can be represented as a (row, column) pair. We typically represent these positions using the simple class `api.Position`, which you will find in the api package. At any stage in the game, a grid position may be empty (null) or may be occupied by an icon (i.e., a colored block), represented by the class `api.Icon`.

In addition, a piece or shape made up of a combination of icons falls from the top of the grid. This is referred to as the current piece or current shape. As it falls, the current piece can be

- shifted from side to side
- transformed, which may rotate it or flip it or something else
- cycled, which will change the relative positions of the icons within the piece, without changing the cells it occupies (that is, without changing the shape and position).

TERMS:

- Game Grid (Rows and columns for the game. Top Left hand corner is row 0 col 0.)
- Position (represents a specific row and column)
- Icon (a color)
- Cell (consists of a position and a color – i.e. Icon)
- Piece (Each piece is in a bounding square. For this assignment all the pieces are of size 3x3. The position of the top-left corner of the bounding square in the game grid is stored by the Piece. The piece has an array of cells. A cell is basically a (x,y) position in the matrix and a color (represented by Icon). Not all coordinates of the bounding box has a cell. See the example for a LShapedPiece in figure 3. The position IN the game grid is 2,3. The bounding square is 3x3. The squares in relative positions (0,0), (0,1), (1,1), (2,1) have cells with colors yellow, cyan, green, and red respectively.

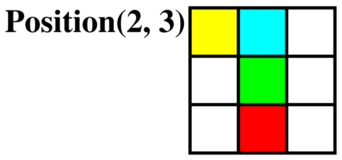

Figure 1: An LShapedPiece

- Invoking `cycle()` (changes color of the CELLS occupied by the piece)
- Invoking `transform()` (changes the position of the CELLS of the piece within the bounding square for the piece).

A user interface is provided which has the following key mappings:

- left arrow - shift the current falling piece left, if possible
- right arrow - shift the current piece right, if possible
- down arrow - make the current piece fall faster
- up arrow - apply the transform() method
- space bar - cycle the blocks within the piece
- 'p' key - pause/unpause

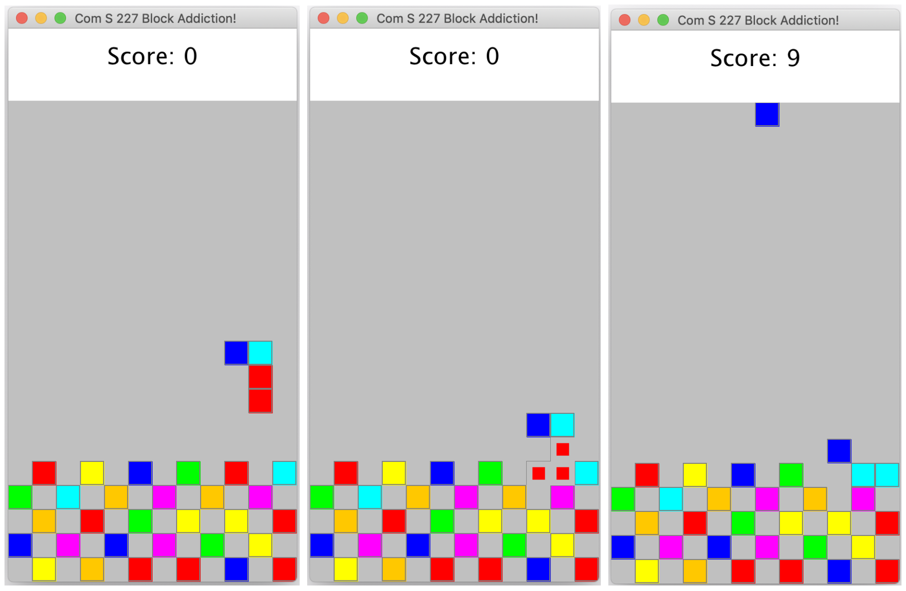
Com S 227 Block Addiction!
Score: 0
Com S 227 Block Addiction!
Score: 0
Com S 227 Block Addiction!
Score: 9

Figure 2: An LShapedPiece falling, and the collapsible set of three matching adjacent red icons disappers, and the blocks above them shift down.

When the currently falling piece can't fall any further, its icons are added to the grid, and the game checks whether it has completed a collapsible set. In BlockAddiction, a collapsible set is formed of three or more adjacent icons that match (have the same color). All icons in a collapsible set are then removed from the grid, and icons above them are shifted down an equivalent amount. (Note that there is no "gravity" and there can be empty spaces remaining below an icon in the grid.) The new icon positions may form new collapsible sets, so the game will iterate this process until there are no more collapsible sets.

This behavior is already implemented in the class AbstractGame, and there is also an abstract method `determinePositionsToCollapse()`, which finds the positions in collapsible sets. You will not have to implement this method – it will be given to you.

SPECIFICATION

The specification for this assignment includes

- this pdf
- the provided javadoc

You will also be provided with some sample code to provide the api you must use as well as some UI code for you to observe your code executing. Note that your code will not be tested using the UI. The UI is just for fun/inspiration. You should depend more on tests for establishing the correctness of your code.

Get the provided source code

The code will be provided to you as a .zip file which contains a whole project missing the files that you need to implement.

- Download and save the file **Homework4Skeleton.zip** file from Canvas
- If your workspace is empty, create a new project in Eclipse called **Homework4tmp**. This is just because Eclipse will not import a zip file when there are no projects in your workspace.
- At the top, choose the **File** menu entry then **Import**.
- Open the **General** option and select **Projects from Folder or Archive File** in the Import dialog and click **Next**.
- Click the **Archive** button to indicate you want to import from an archive. Then browse to choose the zip file from the file system and click **Finish**.
- In the Project Explorer, you can open the newly created project and click on the src directory and you will see directories including api and examples. Be sure to read the file called SamplePiece under examples, and the file Piece under api. Those are the most important files you will be working with. You can create a new package called hw4, and that is where your own files (the six files mentioned above) should be created. You should also create any necessary abstract classes and corresponding test files in there.

The Piece interface and the six concrete Piece types

See the javadoc for the Piece interface. You can generate the javadoc locally by opening the Piece.java file (in the api package), and Click on **Project** menu then **Generate Javadoc**. The generated javadocs will be stored in the doc directory in your project (you must confirm this choice). Then you can browse and search for the Piece.html file and open it in your favorite browser.

The **currently falling piece** is represented by an object that implements the Piece interface.

Each piece has a state consisting of:

- The position in the grid of the upper-left corner of its bounding square, represented as an instance of Position (which can change as a result of calling methods such as shiftLeft(), shiftRight(), or transform()).
- The icons that make up the piece, along with their relative positions within the bounding square.

The position of a piece within a grid is always described by the upper left corner of its bounding square. Most importantly, there is a `getCellsAbsolute()` method that enables the caller to obtain the actual positions, within the grid, of the icons in the piece. The individual icons in the piece are represented by an array of Cell objects, a simple type that encapsulates a Position and an Icon. A Position is just a (row, column) pair, and an Icon just represents a colored block.

For example, one of the concrete Piece classes you will implement is the LShapedPiece. One is shown below in its initial (non-transformed) configuration. The method `getCells()` returns four cell objects with the positions relative to the upper left corner of the bounding square, namely (0, 0), (0, 1), (1, 1), and (2, 1), in that order, where the ordered pairs represent the relative (row, column), NOT the absolute (x, y). (The colors are shown for illustration, and are normally assigned randomly by the generator for the game.) Suppose that the piece's position (upper left corner of bounding square) is row 2, column 3 on the game grid. Then the `getCellsAbsolute()` method should return an array of four cell objects with positions (2, 3), (2, 4), (3, 4), and (4, 4), in that order.

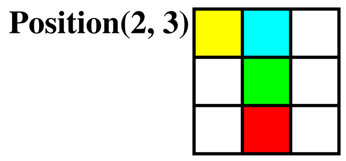

Figure 3: An LShapedPiece

If the method shiftRight() is called on this piece, the position is updated, but `getCells()` would still return the same cells, since the positions are relative to the upper left corner. But the `getCellsAbsolute()` method would now return (2, 4), (2, 5), (3, 5), and (4, 5), as shown in figure 4.

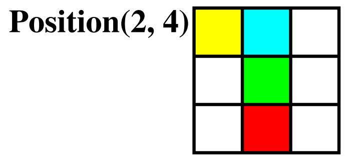

Figure 4: An LShapedPiece

Each piece defines a behavior associated with the `transform()` operation. For the LShapedPiece, the transform() operation just flips it across its vertical centerline. The position of the bounding square does not change, but the positions of the cells within the bounding square are modified. After the modification, the getCells() method should return an array of cells with positions (0, 2), (0, 1), (1, 1), and (2, 1), and the getCellsAbsolute() method should return an array of four cell objects with positions (2, 5), (2, 4), (3, 4), and (4, 4), in that order, as shown in figure 5.

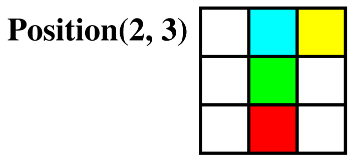

Figure 5: LShapedPiece flipped (reflected) Horizontally

Likewise, if the cycle() method is invoked, the positions for the cells stay the same but the icons associated with the cells will change. The illustration in figure 6 shows the result of invoking cycle() on the initial configuration.

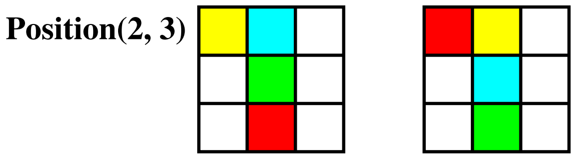

Figure 6: LShapedPiece (left) with icons cycled (right)

Each icon shifts forward to the next cell, and the last icon is placed in the first cell. (The ordering of the cells always the same, even if transformed.)

Altogether you will need to create six concrete classes, described below, that (directly or indirectly) extend AbstractPiece and, therefore, implement the Piece interface. It is up to you to decide how to design these classes, but a portion of your grade will be based on how well you use inheritance to avoid duplicated code. Remember that in the descriptions below, an ordered pair is (row, column), NOT (x, y):

1. The one illustrated above, called the **LShapedPiece**. Initially the icons are at (0, 0), (0, 1), (1, 1), and (2, 1), in that order. The transform() method flips the cells across the vertical centerline.

2. The **TeePiece**, which has a 3 x 3 bounding square, shown below with its initial configuration on the left, with initial icon positions in the order (0, 0), (1, 0), (1, 1) and (2, 0). The transform() method flips the cells across the vertical centerline, as shown on the right (similar to the the LShapedPiece) in figure 7.

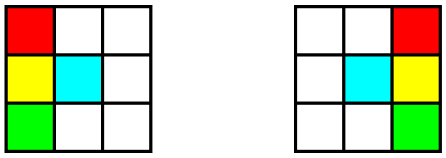

Figure 7: TeePiece (left) after calling transform once (right)

3. The **QuotesPiece**, which has a 3 x 3 bounding square with the icons down the center in the order (0, 0), (1, 0), (0, 2) and (1, 2). This looks like a pair of ticks as in a double quote. For the QuotesPiece, the transform() method rotates the bounding square clockwise by a quarter turn, as shown in figure 8.

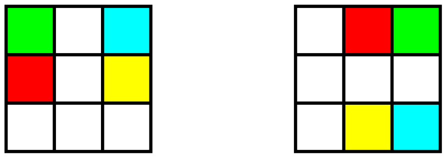

Figure 8: QuotesPiece (left) after calling transform once (right)

4. The **RotatingSPiece**, also with a 3 x 3 bounding square and icons initially in positions (0, 0), (0, 1), (1, 1) and (1, 2). This looks very slightly like an S-shape. For this piece the transform() method rotates the bounding square clockwise by a quarter turn, as shown in figure 9.

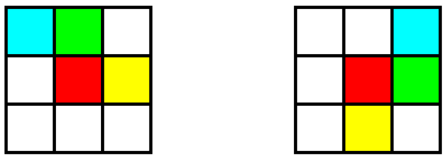

Figure 9: RotatingSPiece (left) after calling transform once (right)

5. The **CirclingPiece**, has a 3 x 3 bounding square and initial cell positions (0, 0), (1, 0), (2, 0) and (2, 1) in that order from the "head" of the snake (or the leading cell) to the tail (the trailing cell). The transform() method moves the piece clockwise around the periphery (edges) of the bounding box like a snake or a train. When three transforms are applied consecutively the resulting sequence is illustrated in figure 10.

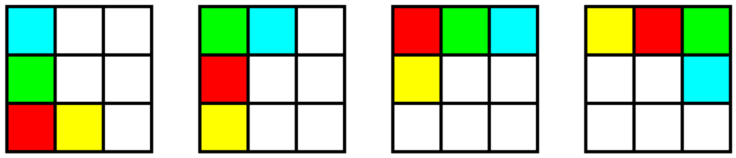

Figure 10: CirclingPiece (left) showing three transforms

Note that it takes eight transforms to arrive back in the original configuration. This is therefore different from rotating the bounding box right by a quarter turn, which would return to its original configuration after four transforms.

6. The **SnakingPiece**, which has a 3 x 3 bounding square and initial cell positions (0, 0), (1, 0), (1, 1) and (1, 2) in that order from the "head" of the snake (or leading cell) to the tail (the trailing cell). The transform() method transitions through twelve different states (like a snake or a train) and back to the original, following a snake-like pattern as shown in figure 11 (reading left-to-right and top-to-bottom):

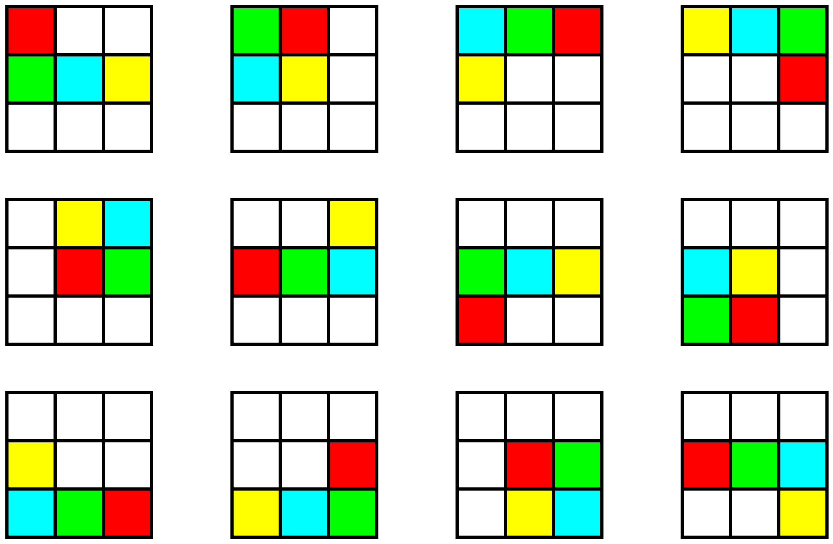

Figure 11: SnakingPiece (top left) showing all twelve states

The motion of the "head" is like drawing a figure of eight (clockwise in the top loop and anticlockwise in the bottom loop) and involves travelling across the middle row twice. Special advice is given below about a relatively easy way to implement this. It is recommended that you implement this first and afterwards you can implement the CirclingPiece while trying to take advantage of similar strategies, and as always trying to reduce code duplication.

When you implement these six concrete types, pay careful attention to code reuse, and implement common code for the Piece interface in an abstract superclass called AbstractPiece. You will also be encouraged to notice similarities in the transform code for specific pieces, which allows you to create their transform functions in abstract classes so that the concrete classes can be a lot shorter / simpler. Part of your score will be based on how well you have taken advantage of inheritance and abstraction to reduce code duplication among the six concrete types.

Each of these types has a constructor that takes an initial position and an array of Icon objects:

```
public LShapedPiece(Position position, Icon[] icons)
public QuotesPiece(Position position, Icon[] icons)
public TeePiece(Position position, Icon[] icons)
public RotatingSPiece(Position position, Icon[] icons)
public CirclingPiece(Position position, Icon[] icons)
public SnakingPiece(Position position, Icon[] icons)
```

The given icon objects are placed in the initial cells in the order given. Each constructor should throw IllegalArgumentException if the given array is not the correct length.

There are some additional notes below about implementing `transform()` for the SnakingPiece and for implementing `clone()`.

Note about the clone() method

TL;DR In most cases you can just copy and paste the clone() method from SamplePiece into AbstractPiece, changing the SamplePiece cast to AbstractPiece.

A Piece must have a clone() operation that returns a deep copy of itself. This is actually a critical part of the implementation, since the mechanism that the AbstractGame uses to detect whether a shift or transform is possible is by cloning the shape, performing the shift or transform first on the clone, and checking whether it overlaps other blocks or extends outside the grid.

The method clone() is defined in java.lang.Object as follows:

```
  protected Object clone() throws CloneNotSupportedException
```

The default implementation of clone() makes a copy of the object on which it's invoked, and although it is declared to return type Object, it always creates an object of the same runtime type. However, it is a "shallow" copy, i.e., it just copies the instance variables, not the objects they refer to. Therefore, in order to use it, you have to add your own code to deep-copy the objects that your instance variables refer to (for the Piece classes, you might have a Position and an array of Cells, for example). In addition, due to some technical nonsense we have to put the call to clone() in a "try/catch" block. We have not seen try/catch yet, but do not worry, you just have to follow the pattern below, as seen in `examples.SamplePiece`:

```
@Override
public Piece clone() {
    try {
        // call the Object clone() method to create a shallow copy...
        SamplePiece s = (SamplePiece) super.clone();
        // ...then make it into a deep copy
        // (note there is no need to copy the position,
        // since Position is immutable, but we have to deep-copy
        // the cell array by making new Cell objects
        s.cells = new Cell[cells.length];
        for (int i = 0; i < cells.length; ++i) {
            s.cells[i] = new Cell(cells[i]);
        }
        return s;
    } catch (CloneNotSupportedException e) {
        // can't happen, since we know the superclass is cloneable
        return null;
    }
}
```

In most cases, you'll implement AbstractPiece using instance variables for the position and cell array as in SamplePiece, and in that case you can just copy and paste the clone() method, changing the SamplePiece cast to AbstractPiece. Basically, you need to make copies of any mutable objects that your instance variables refer to, so that the original and the clone don't share references to the same mutable objects. Immutable objects can be shared when cloning without any problem. So if you have added more reference variables to AbstractPiece, consider whether they need to be copied when you make a clone.

The clone() mechanism in Java tends to be a fragile and often trouble-prone, and it only works on objects that are specifically implemented with cloning in mind. Java programmers tend to avoid using clone() unless there is a good reason. Notice that in the example above, we're not using clone() to make a copy of the the Cell - it's simpler just to use the copy constructor. But for the piece classes, by taking advantage of the built-in clone() mechanism you can implement clone() just once in the superclass and allow the subclasses to inherit it. (Of course, if any of your subclasses have any additional, non-primitive instance variables beyond those defined in your superclass, you'll need to override clone() again for them in order to get a deep copy.)

The Game interface and the AbstractGame class

You don't need to write any code for this section. See the javadoc for the Game interface.

The class AbstractGame is a partial implementation of the Game interface.

YOU DO NOT HAVE TO WRITE CODE FOR THE AbstractGame class. It is PROVIDED to you as well as a concrete implementation of BasicGame.java.

A client such as the sample UI interacts with the game logic only through the interface `Game` and does not depend directly on the `AbstractGame` class or its subclasses. AbstractGame is a general framework for any number of Tetris-style games. It is specialized by implementing the abstract method:

```
List<Position> determinePositionsToCollapse()
```

Examines the grid and returns a list of positions to be collapsed.

(Remember that `List<T>` is the interface type that `ArrayList<T>` implements, so your method can return an `ArrayList<Position>`.)

The key method of Game is step(), which is called periodically by the UI to transition the state of the game. The step() method is fully implemented in AbstractGame, and it is not necessary for you to read it in detail unless you are interested. You will just need a basic understanding of how it interacts with the determinePositionsToCollapse() method that you will implement. In BlockAddiction, the task of determinePositionsToCollapse() is to identify the positions of icons in all "collapsible sets", that is, sets of three or more adjacent icons with matching color.

If you are interested: The way that determinePositionsToCollapse() is invoked from AbstractGame is the following. Whenever the current shape cannot fall any further, the step() method calls determinePositionsToCollapse(). If the returned list is nonempty, then the list is stored in the state variable positionsToCollapse and the game transitions into the COLLAPSING state. On the next call to step(), the method collapsePositions() of

AbstractGame is invoked to perform the modification of the grid to remove the icons and shift down the icons above them. In many of these kinds of games, collapsing some positions may create additional collapsible sets of positions, possibly starting a chain reaction, so the logic of the COLLAPSING state is basically the following:

```
while (game state is COLLAPSING) {
    collapsePositions(positionsToCollapse);
    positionsToCollapse = determinePositionsToCollapse();
    if (positionsToCollapse.size() == 0) {
        // generate a new current shape
        // change game state to NEW_SHAPE
    }
}
```

Remember, the class AbstractGame is fully implemented and you should not modify it.

The BlockAddiction class

YOU DO NOT HAVE TO WRITE CODE FOR THE BlockAddiction class

You will create a subclass of AbstractGame, called BlockAddiction, that implements the game described in the introduction. The method determinePositionsToCollapse() must be declared public even though it is declared protected in AbstractGame (yes, Java allows this). This requirement is to make it easier to test your code.

There are two constructors declared as follows:

```
public BlockAddiction(int height, int width, Generator gen, int preFillRows)
public BlockAddiction(int height, int width, Generator gen)
```

(The second one is equivalent to calling the first one with zero for the fourth argument.) The height, width, and generator are just passed to the superclass constructor. If preFillRows is greater than zero, your constructor should initialize the bottom preFillRows rows in a checkerboard pattern, using random icons obtained from the generator. The checkerboard pattern should place an icon at (row, col) in the grid if both row and col are even, or if both row and col are odd. See the illustration on page 3.

Implementing the method determinePositionsToCollapse()

The method determinePositionsToCollapse() is potentially tricky, and you can waste a lot of time on it if you don't first think carefully. A collapsible set is defined to be any set of three or more adjacent icons with the same color, so it could potentially contain many icons. Given an icon in a collapsible set, it is a hard problem to find just the cells that are part of that set. However, notice that you do not have to solve that problem. You have an easier problem, which is to return a list including all cells that are part of any collapsible set in the grid. What makes a cell part of a collapsible set? Well, either it has two or more neighbors that match its color, or else it must be a neighbor of such a cell. Therefore, it is enough to iterate over the grid and do the following for each array element:

If the element is non-null and has two or more neighbors that match, add it to the list, and also add all its matching neighbors to the list.

The list may contain many duplicates, which is not hard to deal with - just create a new list, copy the positions over, but ignore any that you have already found (the list method contains() is useful here). Finally, the list needs to be sorted. Fortunately, the Position class implements the Comparable interface so you can just call Collections.sort on your list.

The BasicGenerator class

See the javadoc for the Generator interface

YOU DO NOT HAVE TO WRITE CODE FOR THE BasicGenerator class. It is PROVIDED TO YOU.

One important detail for the challenge and playability of the game is to configure what pieces are generated and the probability of getting each type. The role of the Generator interface is to make these features easy to configure. The AbstractGame constructor requires a Generator instance to be provided to its constructor. There is a partially implemented skeleton of the BasicGenerator class implementing the Generator interface. Ultimately, it should return one of the six concrete piece classes, selected according to the probabilities given in the javadoc comments. As you develop your Piece classes, you can just add statements to generate the ones you have completed and tested. (An easy way to generate pieces with varying probabilities: generate a random number from 0 to 100, and if it's in the range 0 up to 10 return an LShapedPiece, if it's in range 11 up to 35 return the next type of piece, and so on.) See the BasicGenerator javadoc comments for more details.

The UI

There is a graphical UI in the ui package. The controls are the following:

- left arrow - shift the current falling piece left, if possible
- right arrow - shift the current piece right, if possible
- down arrow - make the current piece fall faster
- up arrow - apply the transform() method
- space bar - cycle the blocks within the piece
- 'p' key - pause/unpause

The main method is in `ui.GameMain`. You can try running it. It will start up a small instance of the partly implemented class `example.SampleGame`. See the "Getting started" section for more details. To run the UI with your BlockAddiction game once you get it implemented, you will need to edit ui.GameMain.

The UI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course, but of course you are welcome to read it. It is provided for fun, not for testing your code! It is not guaranteed to be free of bugs.

TESTING AND THE SPECCHECKER

As always, you should try to work incrementally and write tests for your code as you develop it. For the Piece hierarchy you can create corresponding files called *something*PieceTests.java and run some basic tests on your own class. Remember for this you will not be submitting your tests, so you are welcome to share your test files with each other (but not your submitted code).

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. In particular, when we grade your work we are NOT going to run the UI, we are going to verify that each method works according to its specification.

We will provide a basic SpecChecker on Gradescope, 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, all the tests will pass.

Remember that your instance variables should always be declared private, and if you want to add any additional methods that are not specified, they must be declared private or protected.

Special documentation and style requirements

- You may not use protected, public , or package-private instance variables. Normally, instance variables in a superclass should be initialized by an appropriately defined superclass constructor. You can create additional protected getter/setter methods if you really need them.
- **Accessor methods should not modify instance variables.**
- Class javadoc must include the `@author` tag, and method javadoc must include `@param` and `@return` tags as appropriate.
  - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
  - **When a class implements or overrides a method that is already documented in the supertype (interface or class) you normally do not need to provide additional Javadoc,** unless you are significantly changing the behavior from the description in the supertype. You should include the @Override annotation to make it clear that the method was specified in the supertype.
- All variable names must be meaningful (i.e., named for the value they store).
- Your code should NOT be producing console output. You may add println statements when debugging, but you need to remove them before submitting the code.
- Internal (//-style) comments are normally used inside of method bodies to explain how something works, while the Javadoc comments explain what a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.) o Internal comments always precede the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.

GETTING STARTED

The following high-level sequence of steps may be a good approach to completing this project.

1. First play the game provided to you in the `tetris.jar` file as described above, if you haven't already. Type on the command line: `java -jar tetris.jar`. Note that the pieces provided in the `tetris.jar` file are mostly different from the ones you will implement.

2. 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. The code you'll implement is mostly specified in the file `api/Piece.java`. Please read it and its javadoc to understand what each method is supposed to do.

3. Next, in the examples package you'll find a simplified, partial implementation of the Piece interface, called SamplePiece, and a simplified, partial subclass of AbstractGame called SampleGame.

   Currently the ui.GameMain class is set up to create another type of game called a BasicGame. Search the code in `ui/GameMain.java` for the words `SampleGame` and enable that, while disabling the creation of a tt BasicGame.

   Then if you run the main class `ui.GameMain`, it will start up the SampleGame game and you should see a red two-block shape falling from the top. Try implementing the shiftLeft and shiftRight methods of SamplePiece to make the block shift from side to side with the arrow keys. That will be a rudimentary Tetris game.

4. There is a better version of a Game class called BasicGame in the hw4 subdirectory. BasicGame will allow you to start implementing and adding your own concrete piece implementations.

   Once you have played with SampleGame to understand it, you can switch `ui.GameMain` to invoke BasicGame instead of SampleGame. Make this change by editing the file `ui/GameMain.java` to comment out the SampleGame creation (search for SampleGame to find it), and instead use the next paragraph after that, which creates a `BasicGame`.

5. General advice about using inheritance: It may not be obvious how to decide what code belongs in AbstractPiece. A good way to get started is to forget about inheritance at first.

   Pick one of the required concrete types, such as `LShapedPiece`, and simply copy the contents of `AbstractPiece` to a new file called `LShapedPiece`, make it **not** abstract, declare that it `implements Piece`. Then search and replace `AbstractPiece` with LShapedPiece *in just that file*, and implement the methods in `api/Piece.java` which are not implemented yet in LShapedPiece.java.

6. When you want to run a new piece, you need to edit `hw4/BasicGenerator.java` to enable it by making your new Piece one of the pieces that the random generator can generate. Initially the random generator will nly generate the red block of two cells (ie. a SamplePiece). Comment out that return statement and write a return statement for an `LShapedPiece` instead.

7. Then, choose another concrete type (the TeePiece may be good to do next), and write all the code for that one the same way you did the previous step. When you have two or more Piece types that can be generated, then you can start using an if-statement in `hw4/BasicGenerator.java` to choose which Piece to generate based on the value of the random number generated. Please read `hw4/BasicGenerator.java` for how your if-statement could look when you have finished the project.

8. At this point you'll notice that you had to write lots of the same code twice. Move the duplicated code into AbstractPiece, and change the declaration for LShapedPiece so it says `extends AbstractPiece` instead of `implements Piece`. Likewise, change the declaration for TeePiece so that it says `extends AbstractPiece`.

That's a good start, and you can use a similar strategy as you implement the other piece types. Note the use of protected constructors for AbstractPiece in order to initialize it. The protected constructors are already done for you.

Note that any code that is shared by all the concrete pieces should be written in the AbstractPiece.

9. Implement some more pieces, this time using the declaration `extends AbstractPiece`. The next two bullet items suggest how to write some of the more difficult pieces like RotatingSPiece and SnakingPiece. It is best to read the suggestions below before implementing those.

10. Implementing transform() for the RotatingSPiece:

    Implementing rotation can be done for any piece by focusing on the background of the piece, that is the whole bounding box, rather than the specific set of occupied cells. To see the coordinate changes that are needed, perform the following exercise:

    - Take a sheet of paper or an index card. It should be rectangular (not square).
    - Write on it the words "Hello World" to remind yourself what the original orientation was.
    - Draw an arrow from the top left corner, downwards along the left edge, about halfway, and label it "i = row".
    - Draw an arrow from the top left corner, rightwards along the top edge, about one-third of the way, and label it "j = col".
    - Draw a square or dot on the paper which is "i" rows down and "j" columns across to the right.
    - Now physically rotate your sheet of paper clockwise by a quarter turn (that is 90 degrees). (If you did this correctly four times you should end up back where you started, but do this only once). Now the long side and short side of the paper should have "exchanged places". The written words "Hello World" should be downwards now if you wrote horizontally to begin with.
    - While staring at the paper in this new orientation note how the new value of i of the square or dot (meaning distance down the new page) depends on the old value of i and j. Also note how the new value of j (that is, the distance to the right across the new page) depends on the old values of i and j.

      This shows you how the old cell at (i, j) will move (that is, the position it will move to) when the clockwise-rotation transform is done. The nice thing is that this is true not just for the square or dot, but for every position with any row i and any column j.
    - Use your conclusions in the previous step to write the transform code.
    - Test it by hand-running a small example to make sure it does what you expect.
    - Adjust your code so that it only rotates the actual cells you have in the Piece shape, rather than the whole background if you haven't done so already.

    Note that when the code is written correctly it is much much shorter than the intuitive description above of why it works.

11. Implementing transform() for SnakingPiece:

    Please do not just write twelve different cases! There are a few ways to approach this, as always, but here is one idea. First, imagine that there is just one cell that starts at (0, 0) and the transform method should shift it to (0, 1), then to (0, 2), then to (1, 0), and so on (picture the red icon in the illustration above). Rather than using a bunch of if- statements, you can just create an array of Position objects in the order you want:

```
private static final Position[] sequence =
{
new Position(0, 0),
new Position(0, 1),
new Position(0, 2),
new Position(1, 2),
new Position(1, 1),
new Position(1, 0),
new Position(2, 0),
new Position(2, 1),
new Position(2, 2),
new Position(1, 2),
new Position(1, 1),
new Position(1, 0),
};
```

and then maintain a counter that goes from 0 to 11 and wraps around back to 0. At each call to transform(), update the counter and set the cell position using the counter as an index into your sequence array. How about the remaining three cells? Notice that the positions you want for them are always the three preceding ones in the sequence array, where, if your index goes *down* from 0 you wrap around to 11 (that is to say, 11 is the correct representation of -1 in this indexing system).

12. As you get further along, you may also notice some other opportunities to create other abstract classes which are shared by two or more Pieces, but not all the Pieces. To express this code in just one place, you should create your own abstract classes in addition to the provided AbstractPiece. You should do this whenever possible to reduce code duplication. Do you notice more than two Pieces which could use the same transform code? Can you create an abstract class to capture that commonality?

13. To run the UI with your BlockAddiction game once you get it implemented, you'll need to edit the create method of ui.GameMain.

IF YOU HAVE QUESTIONS

For questions, please use Piazza. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly. 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.

If you have a question that absolutely cannot be asked without showing part of your source code, post it ONLY in your private channel - 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. 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.

WHAT TO TURN IN

Please submit, on Gradescope, the following six files *and any other abstract classes you created*, all of which must be in the package hw4:

- AbstractPiece.java
- LShapedPiece.java
- QuotesPiece.java
- TeePiece.java
- RotatingSPiece.java
- CirclingPiece.java
- SnakingPiece.java

Please make sure you upload the correct pieces!
Submit the files to Gradescope using the Programming Assignment 4 submission link.