Com S 227
Spring 2015
Assignment 3
300 points

Due Date: Friday, May 1, 11:59 pm (midnight)

There is no "late" deadline for this assignment. All work must be turned in Friday night.

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html , for details.

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. 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.

*Note: Our final exam is Tuesday, May 5. **It will not be possible to have the assignments graded and returned to you prior to the exam**.*

Please start the assignment as soon as possible and get your questions answered right away. We have made the deadline as late as possible to give you some flexibility, but you should plan on getting the assignment done before that!

Introduction

The purpose of this assignment is to give you some experience working with inheritance and abstract classes in a realistic setting.

In this project you will complete the implementation of a simplified version of a Tetris-style or "falling blocks" type of video game. The application is built on an existing framework that includes the UI, a number of interfaces and utility classes, and and an abstract class, **`AbstractBlockGame`**, which contains most of the complicated game logic.

The tasks to be completed by you are briefly summarized below.

1. Create the three concrete classes **`DiagonalPiece`**, **`IPiece`**, and **`LPiece`** that implement the **`IPolyomino`** interface, plus an abstract class **`AbstractPiece`** containing common code for these three types
2. Create the class **`BasicGenerator`** that implements **`IPolyominoGenerator`**
3. Create a concrete subclass **`BlockGame`**, extending **`AbstractBlockGame`**, that implements the game described in this document, that is, implement the methods **`determineCellsToCollapse()`** and **`determineScore()`** as specified

All the code you implement goes in the **`hw3`** package, and the code in the other packages should not be modified (except that you'll need to edit the **`create()`** method of **`GameMain`** to try out your implementation with the GUI).

Description of the game

This particular game, which we'll simply call **`BlockGame`**, was invented for this assignment and is more or less a cross between Tetris and Columns. (If you are not familiar with such games, you might read about them on Wikipedia and find some online versions to try out, but we will carefully define the behavior of everything you need to implement.)

The basic idea is as follows. The game is played on a grid with 24 rows and 12 columns (See Figure 1). Each location in this grid has a position that can be represented as a (row, column) pair. We typically represent these positions using the simple class **`hw3.impl.Position`**. At any stage in the game a grid location may be empty or may be occupied by a colored square, or *block*. In addition, a shape made up of a combination of blocks, called a *polyomino*, falls from the top of the grid. This is referred to as the *current* polyomino. In general the current polyomino can be **shifted** from side to side using the arrow keys, it can be **transformed** using the up-arrow key, which flips it across its vertical axis, and hitting the space bar will **cycle** the blocks within the shape (that is, change the positions of the colors). In addition, the down-arrow can be used to increase the falling speed.

When the currently falling polyomino can't fall any further, its blocks are added to the grid, and the game checks whether it has completed a *collapsible group*. For this game, a collapsible group is any set of three or more adjacent blocks of the same color. (Diagonals are not considered collapsible groups.) All blocks in collapsible groups are removed from the grid, and blocks above them in the same column are allowed to shift down an equal amount. The new block positions may form new collapsible groups, so the game will iterate this process until there are no more collapsible groups. Most of this logic is already implemented in the class **`AbstractBlockGame`** and your main task is to implement the abstract method **`determineCellsToCollapse()`**, which finds the cells in collapsible groups.

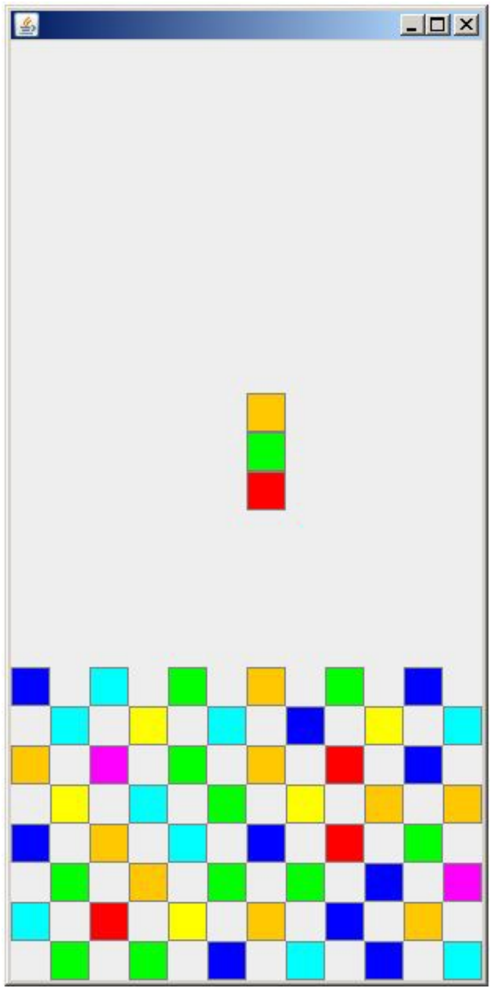


Figure 1 - Example of the initial state of the grid with a falling IPiece

If you have played Tetris before, you might want to note that this game differs from Tetris in the following ways:

- An occupied position in the grid could have empty cells beneath it.
- The game may start out with some of the cells occupied.
- In Tetris, the polyominoes can be rotated 90 degrees at a time; in this game, they can only be flipped side-to-side
- In Tetris, all blocks in a polyomino are the same color, so the "cycle" operation would have no effect
- In Tetris, a "collapsible group" consists of any completely filled row, regardless of the colors of the blocks.

The IPolyomino interface

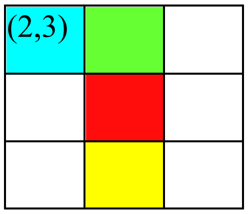
See the javadoc for the **`IPolyomino`** interface.

The currently falling piece, or polyomino, is represented by a concrete type that implements the **`IPolyomino`** interface. Each polyomino has a state including:

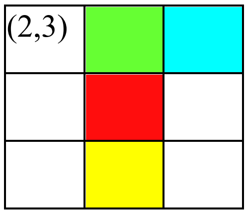
- The position in the grid of its bounding square, represented as an instance of **`Position`**
- The blocks that make up the polyomino, along with their actual locations in the grid (which can change as a result of calling methods such as **`shiftLeft()`**, **`shiftRight()`**, or **`transform()`**).

The position of a polyomino within a grid is always described by the upper left corner of its bounding square. Most importantly, there is a **`getBlocks()`** method that enables the caller to obtain the *actual* locations, within the grid, of the blocks in the polyomino. The individual blocks in the polyomino are represented by the **`Block`** class, a simple type that encapsulates a **`Position`** and a **`Color`** (that is, a constant from the class **`java.awt.Color`**).

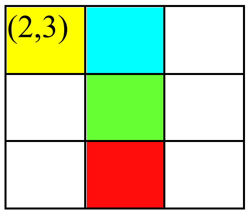
For example, suppose we have **`LPiece`** shown below in its initial (non-transformed) configuration, located in row 2, column 3. (The colors are shown for illustration, and are normally assigned randomly by the generator for the game.) Then the **`getBlocks()`** method should return an array of four block objects with locations (2,3), (2, 4), and (3, 4), and (4, 4), in that order.



Note that if the polyomino in the figure above is transformed using the **`transform()`** method (flipped), the location of the bounding square does not change, but the returned cells will be different, as shown:



Likewise, if the **`cycle()`** method is invoked, the locations for the cells stays the same but the blocks associated with the locations will change. The illustration below shows the result of invoking **`cycle()`** on the first figure:

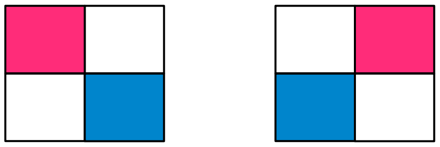


There are three polyomino types to implement:

1. The one illustrated above, called the **`LPiece`**.

2. The **`IPiece`**, which has a 3 x 3 bounding square with the blocks down the center at (0, 1), (1, 1), and (2, 1), as illustrated in Figure 1. (Note that the initial and transformed configurations are identical).

3. The **`DiagonalPiece`**, which has a 2 x 2 bounding square, shown below with its initial configuration on the left and its transformed configuration on the right.



When you implement these three concrete types, pay careful attention to code reuse, and implement common code in an abstract superclass called **`AbstractPiece`**. **Part of your score will be based on how well you have taken advantage of inheritence to reduce code duplication among the three concrete types.**

Each of these types has a constructor that takes an initial position and an array of Color objects.

```
public DiagonalPiece(Position position, Color[] colors)
public IPiece(Position position, Color[] colors)
public LPiece(Position position, Color[] colors)
```

The constructor should throw **`IllegalArgumentException`** if the given array is not the correct length.

Note about the `clone()` method

In order for the framework to work correctly, you need to override the `clone()` method, which creates a duplicate of an object. This method is actually defined as part of `java.lang.Object`, but there are some hoops to jump through to use it correctly. In the skeleton code for `AbstractPiece` you'll find a partial implementation of `clone()`. In the section marked TODO, you need to make *copies* of the objects that your instance variables refer to, so that the original and the clone don't share references to the same objects. There is a more detailed example of doing this in `hw3.example.SamplePiece`. *If you have additional reference variables in the concrete subtypes that are not in* `AbstractPiece`, *you'll need to override* `clone()` *in those classes too*.

The IGame interface and AbstractBlockGame class

The UI interacts with the game through the `IGame` interface. For the most part, this code will not concern you directly, since all the methods are implemented in `AbstractBlockGame`. At some point you might notice that the key method of IGame is `step()`, which is called periodically by the GUI to transition the state of the game. The `step()` method is fully implemented, and it is not necessary for you to read it unless you are interested.

`AbstractBlockGame` is a general framework for any number of Tetris-style games. You will specialize it for this project by extending it and implementing the two abstract methods.

`ArrayList<Position> determineCellsToCollapse()`

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

`int determineScore()`

Returns the current score.

You should not modify `AbstractBlockGame`.

The BlockGame class

You will create a subclass of `AbstractBlockGame`, called `BlockGame`, that implements the game described in the introduction. In particular, you'll need to override `determineCellsToCollapse()` so that it returns all cells that form part of a collapsible group, as described in the introduction. Also, you'll need to override `determineScore()` so that it returns the total number of individual blocks that have been collapsed in the game so far. There are two constructors:

```
public BlockGame(IPolyominoGenerator gen)
public BlockGame(IPolyominoGenerator gen, Random rand)
```

The first form starts with an empty grid. The second form should initialize the grid is as follows: in the bottom 8 rows, alternating locations are filled in a checkerboard pattern, as illustrated in Figure 1 (which also depicts a falling IPiece). The color is selected at random from `AbstractBlockGame.COLORS` using the given `Random` object.

To get access to the grid, use the method `getGrid()`. Note that this method has been declared public to facilitate testing of your `determineCellsToCollapse()` method. One thing you will need to notice is that the grid itself is a 2d array of the type `GridCell`. A `GridCell` has a color and also a boolean *marked* status. Your methods should not modify either of these values. You can easily check whether two cells `c1` and `c2` have the same color by checking `c1.matches(c2)`.

Implementing the method `determineCellsToCollapse()`

The method `determineCellsToCollapse()` is potentially tricky, and you can waste a lot of time on it if you don't first think carefully. A collapsible group is defined to be any set of three or more adjacent cells with the same color, so it could potentially contains many cells. Given a cell in a collapsible group, it is a hard problem to find *just* the cells that are part of that group. 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 group. What makes a cell part of a collapsible group? Well, either it has two or more neighbors that match its color, or else it must be a neighbor of such a cell (otherwise it would be a group of just two cells). So it is enough to iterate over the list and do the following:

> *If the cell 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 duplicates, which is not a problem. (However, you may need to eliminate duplicates in order to correctly update the score).

The BasicGenerator class

See the javadoc for the `IPolyominoGenerator` interface.

When constructing an instance of `BlockGame`, you need to supply an instance of `IPolyominoGenerator`. This will be used by the game each time it needs to create a new shape dropping from the top. One of your tasks is to write a `BasicGenerator` class implementing `IPolyominoGenerator`. It is pretty simple. It is constructed with an instance of `Random`, and the class uses that `Random` object to select the concrete type of `IPolyomino` to return and to randomly select the colors. You should return the three types with the following probabilities:

`IPiece`	.60
`DiagonalPiece`	.20
`LPiece`	.20

In all cases the colors of the blocks should be assigned uniformly at random from the array `AbstractBlockClass.COLORS.` Duplicate colors are allowed in a polyomino.

The initial position of the bounding square is specified for each type as follows:

`IPiece` :	(-2, 5)
`DiagonalPiece`	(-1, 5)
`LPiece`	(-2, 5)

Importing the sample code

The sample code is an archive containing a complete Eclipse project that you can import. See the instructions from homework 2 (page 9 of that pdf) if you don't remember how to do this.

The GUI and the sample game

As an example, the `SampleGame` (in package `hw3.example`) class is an extension of `AbstractBlockGame` implementing a simple form of Tetris. That is, `determineCellsToCollapse()` returns all cells that form a completed row, and `determineScore()` returns the total number of rows completed so far in the game. There is also a partially written implementation of `IPolyomino` called `SamplePiece`, and a very simple generator called `SampleGenerator`. If you run `hw3.ui.GameMain`, you should see the simple two-block piece falling from the top. If you then implement the `shiftLeft()` and `shiftRight()` methods of `SamplePiece`, you can play a really basic version of Tetris.
In order to run the GUI with your own game, edit the first few lines of the `create()` method of `GameMain`. If you want to be able to reproduce the results, you can seed the generator.

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. *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, 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 specfications, 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 the following message in the console output:

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

where x is some number. This SpecChecker will also offer to create a zip file for you that will package up the six 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`** or possibly **`protected`**.

See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

Getting started

See the sections above "Importing the sample code" and "The GUI and the sample game". The code should compile out of the box and you should be able to run the sample game.

Your implementation of **`BlockGame`** is not dependent on having the **`IPolyomino`** classes implemented, so you can develop and test it independently of everything else. However, you won't be able to implement **`BasicGenerator`** until you have completed the three concrete subtypes of **`IPolyomino`**.

Be careful with your import statements for **`Position`**. Be sure you import **`hw3.api.Position`**, not **`javax.swing.text.Position`** (which might be suggested by the Eclipse "quick fix").

A good place to begin is to experiment with the **`SamplePiece`**. Read the code and be sure you understand it. Then try writing the **`shiftLeft()`** and **`shiftRight()`** methods as a kind of a warm-up. You can use similar ideas when you implement your own, if you wish.

When you are implementing a class hierarchy like **`DiagonalPiece`**, **`IPiece`**, and **`LPiece`** and **`AbstractPiece`**, it is not always easy to tell at first what code should go into the abstract superclass. A good way to begin is to choose just one of the concrete types to start with, say **DiagonalPiece**. Implement it and test it completely without worrying about **AbstractPiece**. Then, when you start writing the next one, say **IPiece**, you will notice you are writing some of the same code again. That is a good clue about what can go in the abstract superclass.

As always, you should start by making up some test cases or usage examples to be sure you know what the code is supposed to do. When writing tests it is useful to notice that **Block** and **Position** override the **toString** method to provide a readable representation. For example, you could do something like this to check whether the block positions are being returned correctly:

```
public static void main(String[] args)
{
  Color[] colors = new Color[2];
  colors[0] = Color.RED;
  colors[1] = Color.BLUE;
  DiagonalPiece d = new DiagonalPiece(new Position(2, 5), colors);
  Block[] blocks = d.getBlocks();
  System.out.println(Arrays.toString(blocks));
  System.out.println("Expected [red at (2, 5), blue at (3, 6)]");
  d.shiftDown();
  System.out.println(d.getPosition());
  System.out.println("Expected (3, 5)");
  blocks = d.getBlocks();
  System.out.println(Arrays.toString(blocks));
  System.out.println("Expected [red at (3, 5), blue at (4, 6)]");
  d.transform();
  System.out.println(d.getPosition());
  System.out.println("Expected (3, 5)");
  blocks = d.getBlocks();
  System.out.println(Arrays.toString(blocks));
  System.out.println("Expected [red at (3, 6), blue at (4, 5)]");
}
```

(See the top of AbstractBlockGame to see what abbreviations are being used for the colors.)

Before you start implementing **AbstractBlockGame**, read the section above entitled "Implementing the method **determineCellsToCollapse()**" and work out on paper how you are going to try to do it. For testing, use the fact that there is a public **getGrid()** method in **AbstractBlockGame**, so you can create a game with empty grid and then set values in the grid for testing. For example,

```
BlockGame game = new BlockGame(new SampleGenerator());
GridCell[][] grid = game.getGrid(); // reference to the actual game array
grid[2][3] = new GridCell(Color.RED);
grid[3][3] = new GridCell(Color.BLUE);
grid[2][4] = new GridCell(Color.RED);
grid[3][4] = new GridCell(Color.RED);
ArrayList<Position> cells = game.determineCellsToCollapse();
System.out.println(cells);
System.out.println("Expected [(2, 4), (3, 4), (2, 3)] in some order");
```

In the default package of the sample code you'll also find a short example including the code above and also a method `printGrid()`, that will print out the grid using a one-letter abbreviation for the color in each cell.

Style and documentation

Special notes regarding this assignment

- Part of the purpose of this assignment is for you to think about the use of inheritance as a design strategy for reducing duplicated code. In most designs a small amount of duplication is unavoidable, but strive to keep it to a minimum while maintaining logical consistency.
- Do not use `instanceof` or other hacks to explicitly check the type of an object.
- When you are implementing a method that is defined in an interface, or overriding a method that was defined in a superclass, it is normally NOT necessary to rewrite the javadoc, unless you have substantially changed its meaning. Please use the @Override annotation, however, to make it clear that the method was originally defined in a supertype.
- Do not use `public` or `protected` variables. If you need to initialize a variable in a superclass, do so by calling the superclass constructor. If you need to access a variable in a superclass, implement a `protected` accessor method.
- Do not use "package-private" access anywhere (this is where you leave off the `private`, `protected`, or `public` modifier).

Roughly 15% of the points will be for documentation and code style. The general guidelines are the same as in Homework 1. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each class you write or edit
- You must javadoc each class, instance variable and helper method that you add, except for methods you are overriding that are documented in the interface or supertype
- 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 `assignment3`. 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 `assignment3`. *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 "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "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 posts from the instructors on Piazza 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 Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the SpecChecker. The file will be named `SUBMIT_THIS_hw3.zip`. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, `hw3`, which in turn contains six files:

```
hw3/AbstractPiece.java
hw3/BasicGenerator.java
hw3/BlockGame.java
hw3/DiagonalPiece.java
hw3/IPiece.java
hw3/LPiece.java
```

Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Blackboard using the Assignment 3 submission link and **verify** that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

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 six files listed above. Make sure you are turning in **`.java`** files, not the **`.class`** files. You can accomplish this easily by zipping up the **src** directory of your project. (The zip file will include the other **`.java`** files in the project, but that is ok.) 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.