Com S 227
Summer 2022
Assignment 3
300 points

Due Date: Friday, July 8, 11:59 pm (midnight)

General information

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. Please start the assignment as soon as possible and get your questions answered right away!

Introduction

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.

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by how effectively you have been able to use inheritance to minimize duplicated code in the Piece hierarchy.

Summary of tasks

1. Create implementations of five concrete subtypes of the `Piece` interface:
   - `CornerPiece`
   - `DiagonalPiece`
   - `IPiece`
   - `LPiece`
   - `SnakePiece`
2. Implement the abstract class `AbstractPiece` containing the common code for the concrete `Piece` classes above.
3. Implement a class `BlockAddiction` extending the `AbstractGame` class, in which you provide implementations of the method `determinePositionsToCollapse()` and the constructors.
4. Finish the implementation of the `BasicGenerator` class

All your code goes in the package `hw3`.

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.

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 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

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 **`AbstractBlockGame`**, and your main task is to implement the abstract method **`determineCellsToCollapse()`**, which finds the positions in collapsible sets.

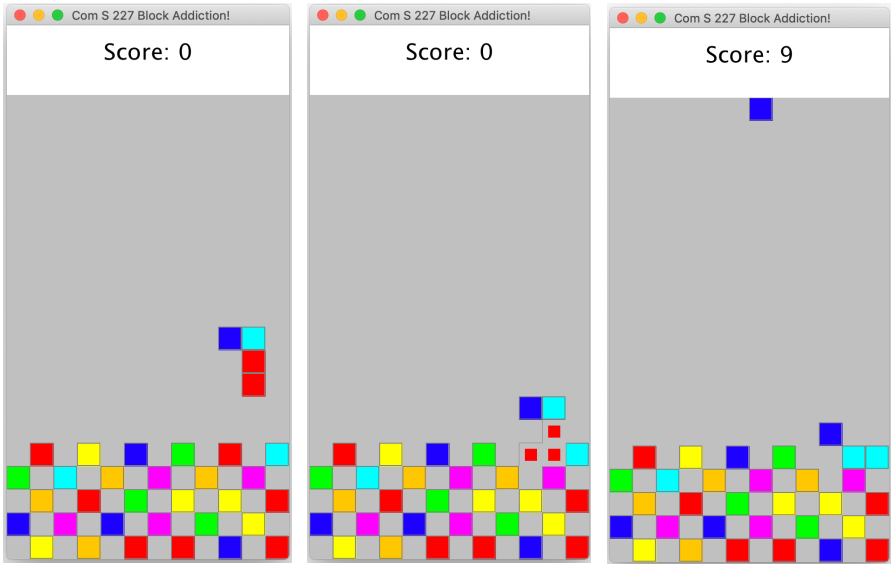


Figure 1: *An LPiece falling, and the collapsible set of three matching adjacent red icons disappears, and the blocks above them shift down.*

Specification

The specification for this assignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

The `Piece` interface and the five concrete `Piece` types

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

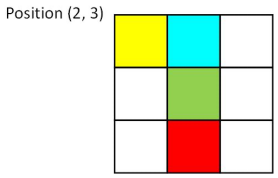
The currently falling piece is represented by an object that implements the **`Piece`** interface. Each piece has a state consisting of:

- 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.

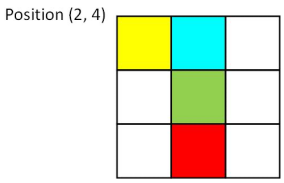
The position of a piece within a grid is always described by the upper left corner of its bounding square. Most importantly, there is a **`getCellsAbsolute()`** method that enables the caller to obtain the *actual* positions, within the grid, of the icons in the piece. The individual icons in the piece 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.

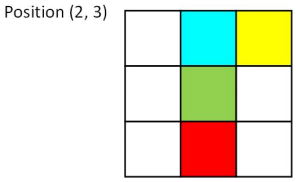
For example, one of the concrete **`Piece`** classes you will implement is call the **`LPiece`**. One is shown below in its initial (non-transformed) configuration. The method **`getCells()`** returns four cell objects with the positions relative to the upper left corner of the bounding square, namely (0, 0), (0, 1), (1, 1), and (2, 1), in that order, where the ordered pairs represent *(row, column)*, **NOT** *(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. 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.



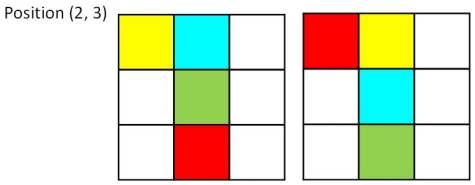
If the method **`shiftRight()`** is called on this piece, the position is updated, but **`getCells()`** would still return the same cells, since the positions are relative to the upper left corner. But the **`getCellsAbsolute()`** method would now return (2, 4), (2, 5), (3, 5), and (4, 5).



Each piece defines a behavior associated with the **`transform()`** operation. For the **`LPiece`**, 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, as shown below. This time 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.



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 below shows the result of invoking **`cycle()`** on the first figure:

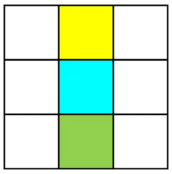


Each icon shifts forward to the next cell, and the last icon is placed in the first cell. (The ordering of the cells always the same, even if transformed.)

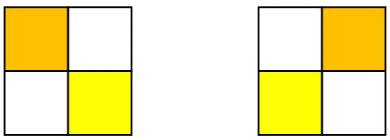
Altogether you will need to create five concrete classes, described below, that (directly or indirectly) extend **`AbstractPiece`** and, therefore, implement the **`Piece`** interface. It is up to you to decide how to design these classes, but a portion of your grade will be based on how well you use 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 **`LPiece`**. 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 **`IPiece`**, which has a 3 x 3 bounding square with the icons down the center in the order (0, 1), (1, 1), and (2, 1). For **`IPiece`**, the **`transform()`** method does nothing:

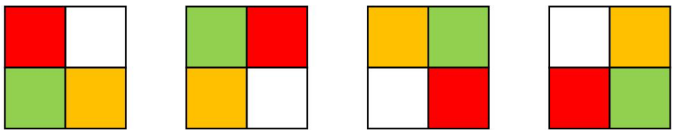


3. The **`DiagonalPiece`**, which has a 2 x 2 bounding square, shown below with its initial configuration on the left. The icon positions are in the order (0, 0), (1, 1). The **`transform()`** method flips the cells across the vertical centerline, as shown on the right (same as for the **`LPiece`**).

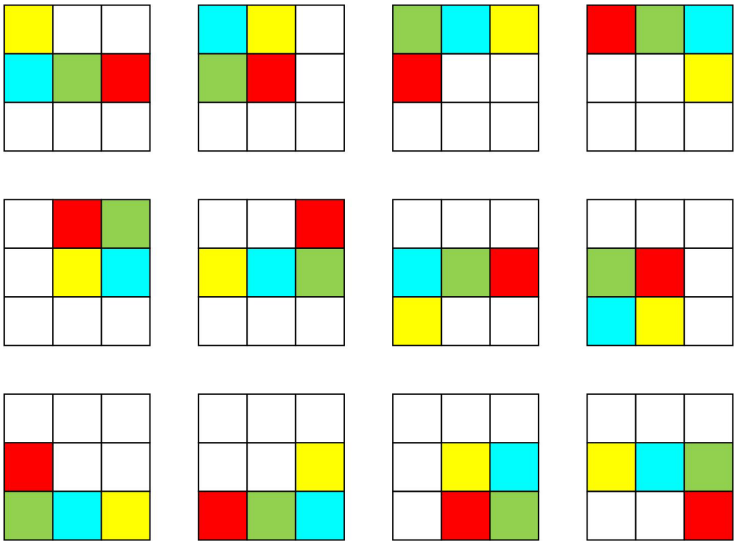


4. The **`CornerPiece`**, which has a 2 x 2 bounding square and initial cell positions (0, 0), (1, 0), and (1, 1) in that order. The **`transform()`** method performs a kind of rotation through four different states as

shown below, e.g., after calling `transform()` once, the cell positions are (0, 1), (0, 0), and (1, 0), and after calling `transform()` four times they are back in the initial positions.



5. The `SnakePiece`, which has a 3 x 3 bounding square and initial cell positions (0, 0), (1, 0), (1, 1), and (1, 2) in that order. The `transform()` method transitions through twelve different states and back to the original, following a snake-like pattern as shown below (reading left-to-right and top-to-bottom):



When you implement these five 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 inheritance 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 `Icon` objects:

```
public CornerPiece(Position position, Icon[] icons)
public DiagonalPiece(Position position, Icon[] icons)
public IPiece(Position position, Icon[] icons)
public LPiece(Position position, Icon[] icons)
public SnakePiece(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 **`SnakePiece`** and for implementing **`clone()`**.

The Game interface and AbstractGame class

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

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

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 class **`BasicGenerator`** class implementing the **`Generator`** interface. Ultimately, it should return one of the five 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 **`LPiece`**, if it's in range 11 up to 35 return a **`DiagonalPiece`**, 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 follow the examples in ExamplePieceTests.java in the sample 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.* **In the default package in the sample code there are some examples with ideas for how to test.**

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 methods that are not specified, they must be declared **`private`** or **`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.

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.

Style and documentation

See the special requirements above.

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
  - You will lose points for having lots of unnecessary instance variables
  - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. 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.)
  - Internal comments always *precede* the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.
  - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

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 "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 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.

Getting started

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. That is, pick one of the required concrete types, such as **`LPiece`**, add the clause **`implements Piece`** to its declaration, and just write the code for all the specified methods. (That is, temporarily forget about the requirement to extend **`AbstractPiece`**.) Then, choose another concrete type, and write all the code for that one. 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 **`LPiece`** so it says **`extends AbstractPiece`** instead of **`implements Piece`**.

That's a good start, and you can use a similar strategy as you implement the other piece types. Remember you'll need to include one or more protected constructors for **`AbstractPiece`** in order to initialize it.

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. There are some example tests provided with the Piazza homework links to give you some ideas about how to test your code. The comments should explain everything pretty well. You can work independently on the **`Piece`** classes and on the **`BlockAddiction`** class. The **`BasicGenerator`** is not directly needed until you want to actually try to play the game.

In addition, in the **`examples`** package you'll find a simplified, partial implementation of the **`Shape`** interface, called **`SamplePiece`**, and a simplified, partial subclass of **`AbstractGame`** called **`SampleGame`**. If you run the main class **`ui.GameMain`**, it will start up this game and you should see a 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.

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

Implementing transform() for `SnakePiece`

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 yellow 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 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.

(And here is an interesting challenge for you: could you reuse this same strategy for **`CornerPiece`**?)

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 **`Shape`** classes, you might have a **`Position`** and an array of **`Cell`**s, 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.SampleShape`**:

```
@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 objects. So if you have added more reference variables to **`AbtractPiece`**, 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.)

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 a minimum of eight files, including:

```
AbstractPiece.java
BasicGenerator.java
BlockAddiction.java
CornerPiece.java
DiagonalPiece.java
IPiece.java
LPiece.java
SnakePiece.java
```

(plus any other abstract class you have added to the hierarchy).

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. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found on Canvas.

We recommend that you submit the zip file as created by the specchecker. If necessary, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the the required .java files. You can accomplish this by zipping up the **src** directory within 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.