Omino

- things commonly omitted
  - *tester* – `OminoTester` should contain test cases for `Polyomino`, `Piece`, `Board` as specified in the handout

Testing:

Like `Block`, `Polyomino` mostly just holds information and provides access to it. There are a couple of things worth testing, though.

Add tester subroutines and test cases for `getNextRotation` and `getBlocks` to `OminoTester`. For `getBlocks`, though, there's a wrinkle — checking that an array of `Block`s has the right contents is a bit cumbersome. Address this by adding a `private` helper method `blocksToString` to `OminoTester`. `blocksToString` should take an array of `Block`s and return a `String` formatted like the `String`s given to `Polyomino` (i.e. in the form `r1 c1 r2 c2 r3 c3 ...`). You can then use `blocksToString` to make a `String` version of the `Block`s array for easier comparison with the expected result.

  - *comments* – Javadoc-style comments for classes and methods
  - *preconditions* – consider preconditions for all methods
    - identify them in the method comments and check them in the method body

Omino

- be sure to follow the specifications in the handout
  - include all of the methods specified, named as directed (when a name is given) and with the parameters specified
    - no reason to change the order of the parameters from how they are listed

- `canPlace`, which takes a piece and its current position (row and column) as parameters and returns whether it is possible to add the piece to the board in that position
- `addPiece`, which takes a piece and its current position (row and column) as parameters and adds the piece to the board

Omino

inappropriate information. Also choose descriptive names and follow consistent and standard conventions for naming and whitespace. It is strongly recommended that you follow the 225 programming standards. Autoformat will take care of many potential whitespace-related issues, including improper indentation and too-long lines. Use blank lines for grouping and organization within longer methods.

- naming conventions
  - end instance variable names with _
  - constants in ALL CAPS
  - name boolean methods so that the code reads well in an `if` statement
    - compare `if ( board.getStatus(row,col) ) { … }` to `if ( board.isOccupied(row,col) ) { … }` or `if ( board.hasPiece(row,col) ) { … }` or `if ( board.pieceAt(row,col) ) { … }`
    - convention is `isXYZ` but other readable versions are OK

Omino

- There are two versions of `Color` — you want the JavaFX version. Review the `import` statements at the very beginning of each class that involves `Color` (`Polyomino`, `Piece`, etc) — if you see `import java.awt.Color;`, delete it and either replace it with `import javafx.scene.paint.Color;` or choose the import involving a javafx package from the auto-fix suggestions.

- remove any `import java.awt.Color;` statements and replace with `import javafx.scene.paint.Color;`

`Board`

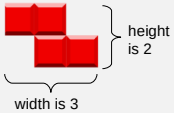
`Board` is the main playing area where the pieces land. It is represented as a grid of squares with (0,0) in the lower left corner, and keeps track of which pieces the blocks occupying squares come from.

- (0,0) for the board should be in the lower left corner, not the upper left corner
  - rows get smaller as pieces move down the board

Omino

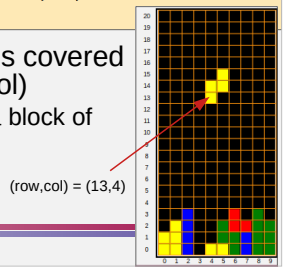
- `computeWidth`, which takes an array of `Block`s and returns the width
- `computeHeight`, which takes an array of `Block`s as a parameter and returns the height

- compute the dimensions of the piece in its current orientation
  - (not the number of blocks / length of the array)



- `canPlace`, which takes a piece and its current position (row and column) as parameters and returns whether it is possible to add the piece to the board in that position
- `addPiece`, which takes a piece and its current position (row and column) as parameters and adds the piece to the board

- check/update all of the board positions covered by blocks of the piece, not just (row,col)
  - (row,col) might not even be covered by a block of the piece



Omino

- robustness
  - printing a message is not appropriate error-handling if the error stems from values coming from outside the method
    - throw `IllegalArgumentException` for violated preconditions
  - make sure the program doesn't crash if the user tries to move a piece off the side of the board
- instance variables and helper methods should be `private`
- constants should be `static` (and `final`)

```
private static final String[][] POLYOMINOES =
  { { "0 0  1 0  2 0", "0 0  0 1  0 2" },
    { "0 0  1 0  0 1", "0 0  0 1  1 1", "1 0  0 1  1 1", "0 0  1 0  1 1" } };
```


Omino

- it can be awkward to have to preemptively undo something – instead only update it when needed

```
for ( int i = 0 ; i < n ; i++ ) {
  if ( … ) {
    …
    i--;
  }
}

for ( int i = 0 ; i < n ; ) {
  if ( … ) {
    …
  } else {
    i++;
  }
}
```