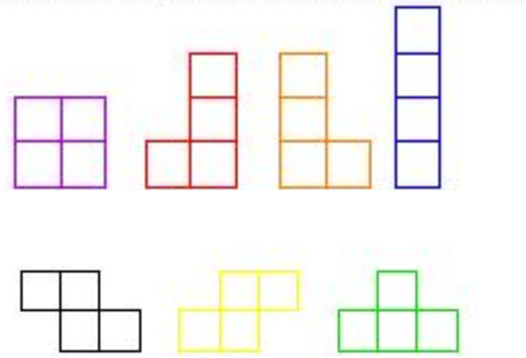
Tetris Help Session

- Game Overview
- Piece Generation
- User Interaction
- Moving and Rotating Pieces
- Maintaining the Board
- Design and Roadmap

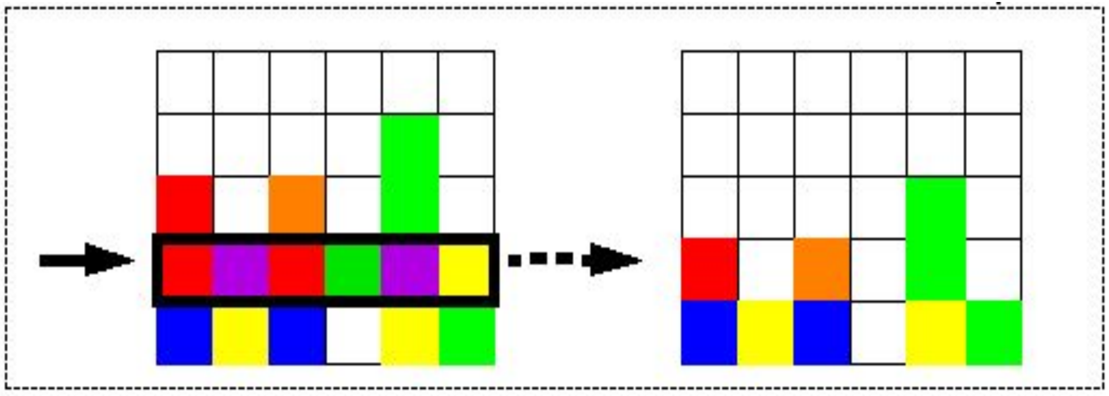
Tetris Game Overview (1/3)



- Tetris pieces move down the board.
  - One square at a time, at regular intervals.
  - Only one piece should be moving at any given time.
- A piece is made up of four squares.
- The user can make the current piece move left, right, down, rotate, and drop.
- The user can pause the program using the keyboard.
- No two squares can occupy the same place on the board.
  - A piece cannot move if any of its squares would move into a space that is already occupied.

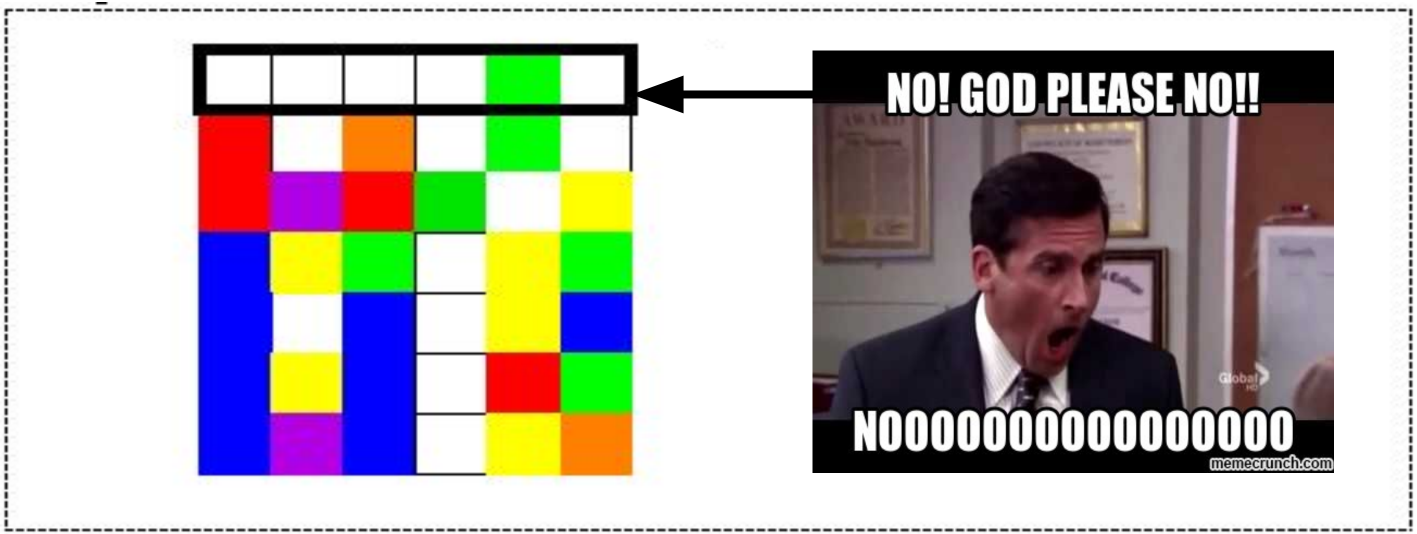
Tetris Game Overview (2/3)

- After a piece cannot fall any further:
  - Its squares become part of the board.
  - A new piece starts falling from the top.
- When a row gets filled, it should disappear, and all the rows above it should “fall” down by one.



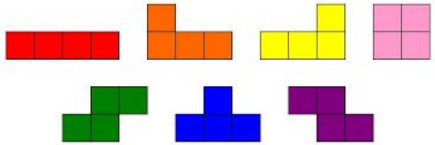
Tetris Game Overview (3/3)

- If the top row of the board has a square in it, or a new piece has no room to fall, then the game is over!



`Piece` Generation

[0]
[1]
[2]
[3]
[ARRAY LENGTH = 4]



- There are 7 different types of Tetris **`Pieces`**
- How can we generate 7 distinct types of pieces?
- Two ways of thinking about it:
  - 7 pieces → 7 different piece classes
    - Could factor out common code to a parent class
    - May need to override methods (see [Design Patterns]() lecture for why this might be dangerous)
    - Is this good use of inheritance?
  - **7 pieces → 7 different configurations of 4 squares**
    - Create one **`Piece`** class, and come up with a way to model the different configurations within that class
    - **We recommend using this implementation!**

`Piece` Generation - Layout

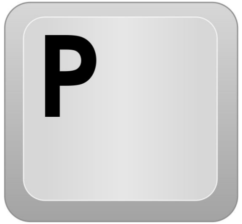
- What is the best way to model the configuration of pieces so that we can use the same technique for each piece to set the **initial positions** of squares?
- Can you think of a way to efficiently store and access 4 squares to make 7 different pieces?
- How to make & store **coordinates** of squares?
  - Store coordinates for the 4 squares of each piece shape in the same type of data structure?
    - 4x2 2D array: 4 squares, 2 coords (x and y) for each one
    - Remember: can initialize an array as int[] myArray = {1, 2, 3};
      - called “static array initialization”
    - Allows us to use the same code to configure each piece shape
    - Where should you put this?
      - Hint: Do your coordinates ever change?
- There are other ways to do this!

`Piece` Generation - Random Pieces

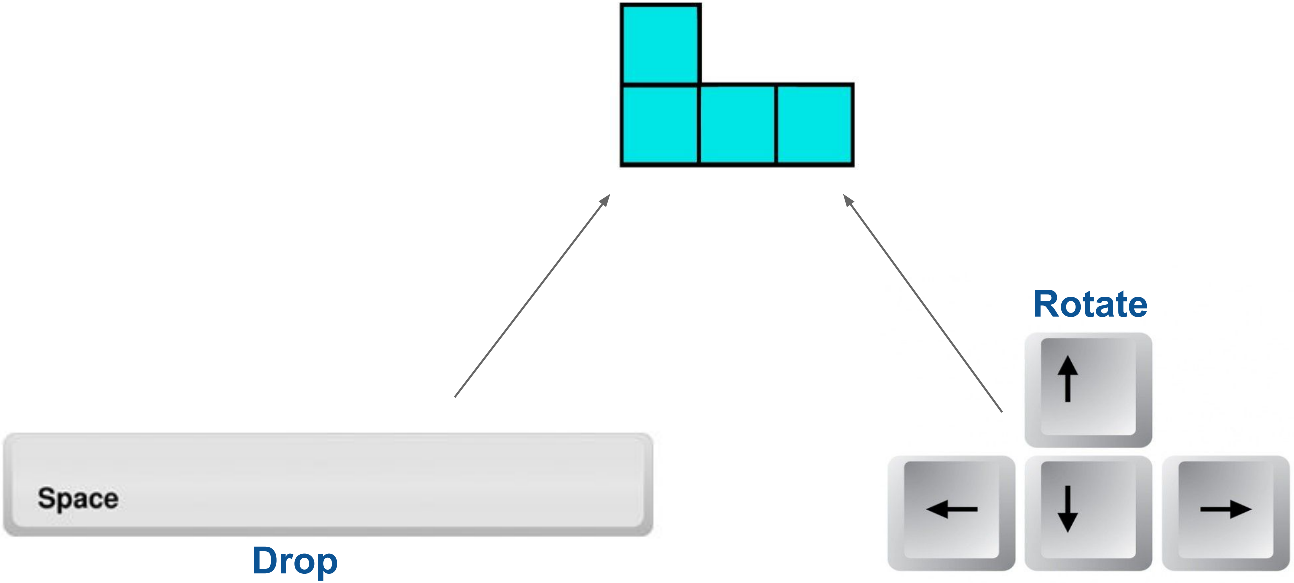
- You want to create bunches and bunches of **`Piece`**s, over and over again.
- Use the Factory Pattern!
    - It has the ability to create new objects.
    - Remember this from lecture? (**`generatePaper()`**)
    - It can be a class with a method, or simply a method which returns a new random **`Piece`**
        - Remember... **`Math.random()`**?

See the Making Decisions Lecture for more information on the Factory Pattern and the **`switch`** statement.

User Interaction (Keyboard)



Pause/unpause game



Drop

Rotate

Move left

Move down

Move right

User Interaction (Keyboard)

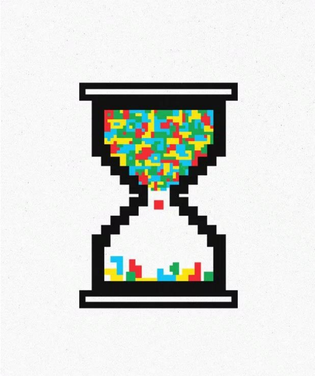
Remember!

- Implement a **`javafx.event.EventHandler<KeyEvent>`**
  - call **`consume()`** method on the KeyEvent at the end

- Call **`requestFocus()`** and **`setFocusTraversable(true)`** in the Pane that listens to your KeyEvents
- Call **`setFocusTraversable(false)`** on other **`Node`**s

Other useful links: Event processing, JavaFX Guide, Doodle Jump Handout

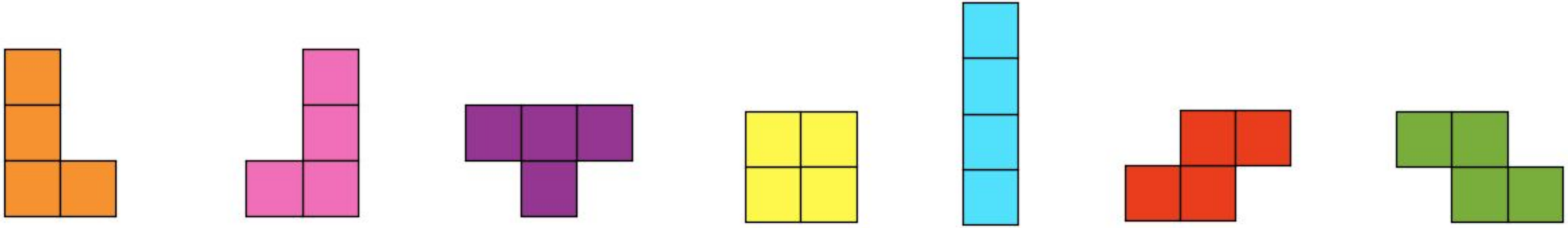
Moving a `Piece` (1/2)



- We know a `Piece` is made up of four squares.
- How do I move a `Piece`?
  - Move all of its squares
  - Or more accurately, have the piece tell all of its squares to move themselves...
    - Change location with `setX()`, `setY()`
- Remember to use a `Timeline` to animate your `Piece` going down!
- Make sure to keep your `move` and `rotate` methods within your `Piece` class!

Moving a `Piece` (2/2)

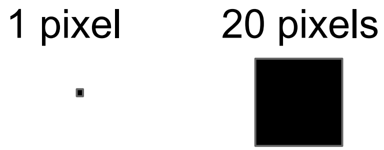
- Can a `Piece` always complete its move?
  - No!
- How do I know if a `Piece` can move?
  - If all four of its squares can move
- How do I know if a square can move?
  - If the place it wants to move to on the board is empty
  - ***And*** if the place it wants to move to is actually on the board!

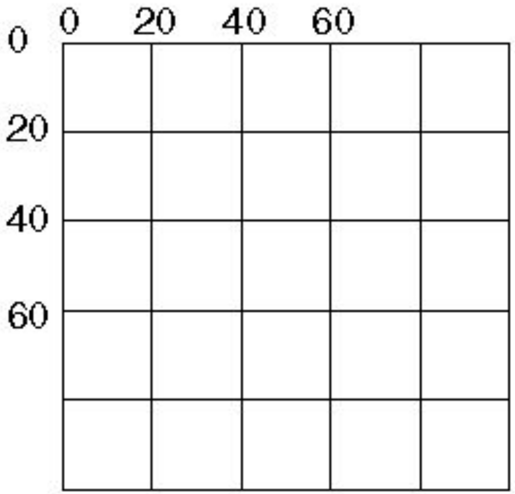


Converting from Indices to Pixels

- The squares’ unit of size is in pixels
  - Pixels are very small.
  - Squares should be 20 pixels by 20 pixels or more.
- I feel a `Constant` comin’ on!
  - Make a `Constants` class, with constants for `SQUARE_SIZE, NUM_ROWS`, etc.
    - Remember to also include `BOARD_WIDTH`, `BOARD_HEIGHT`, etc.
- So to set the location of a square:

```
x = col * Constants.SQUARE_SIZE
y = row * Constants.SQUARE_SIZE
```





Rotate (1/2)

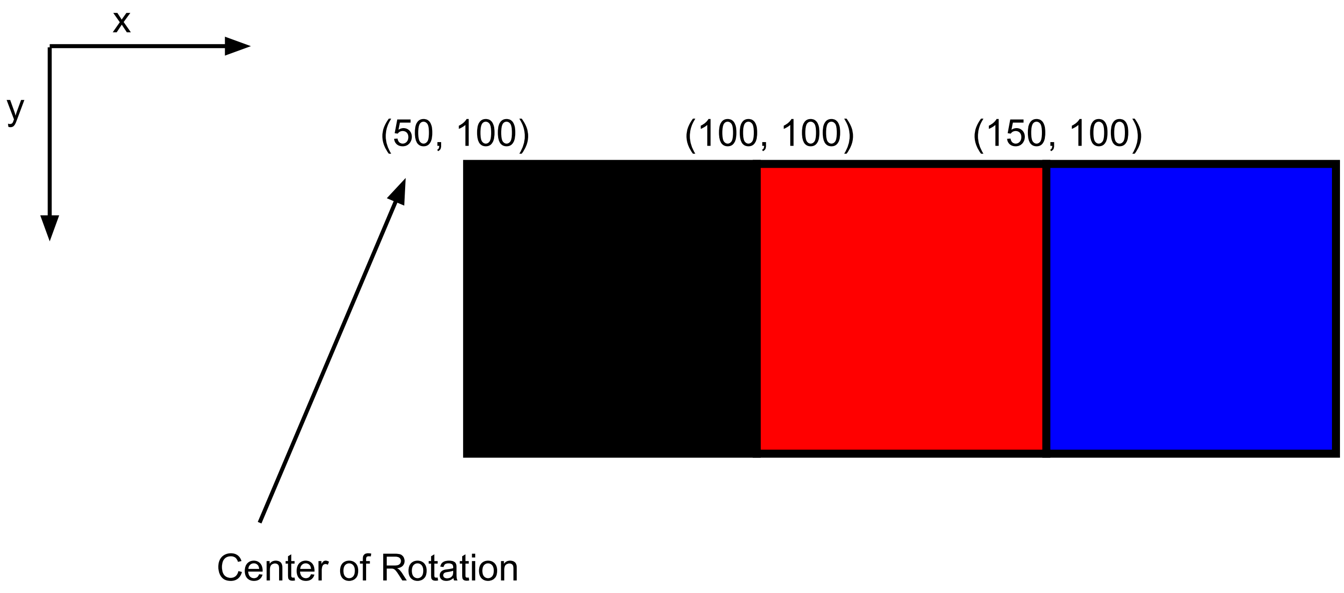
- How do I rotate a point 90 degrees around another point?

```
// Set to the value of x and y of the center point around which I am rotating
int centerOfRotationX;
int centerOfRotationY;
// Set to the value of the point’s current position’s x and y
int locX;
int locY;
// Calculate coordinates of the rotated point
int newXLoc = centerOfRotationX – centerOfRotationY + locY;
int newYLoc = centerOfRotationY + centerOfRotationX – locX;
```

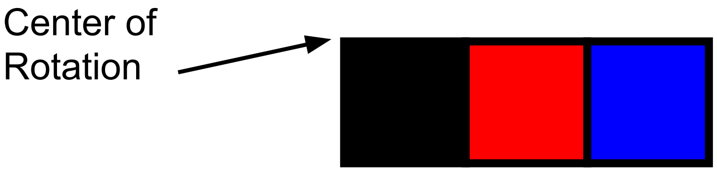
Rotate (2/2)

- Notice that we need to know about the center of rotation
  - the `Piece`’s “center"
  - <u>**Note**</u>: the "center" doesn't necessarily mean the "geometric center" of the piece, it is the point around which we are rotating
    - it is the **top left corner** of *one of the squares* of the `Piece`.
- What does this formula do?
  - It rotates one square around another square.
  - Let’s rotate a red square and a blue square around a black one!

Graphical Rotation Example (1/3)



Graphical Rotation Example (2/3)



Red Square

Center of Rotation (CoR) = (50,100)

Old Location (OL) = (100, 100)

New X = CoRx - CoRy + OLy
= 50 - 100 + 100 = **50**

New Y = CoRx + CoRy - OLx
= 50 + 100 - 100 = **50**

Blue Square

Center of Rotation (CoR) = (50,100)

Old Location (OL) = (150, 100)

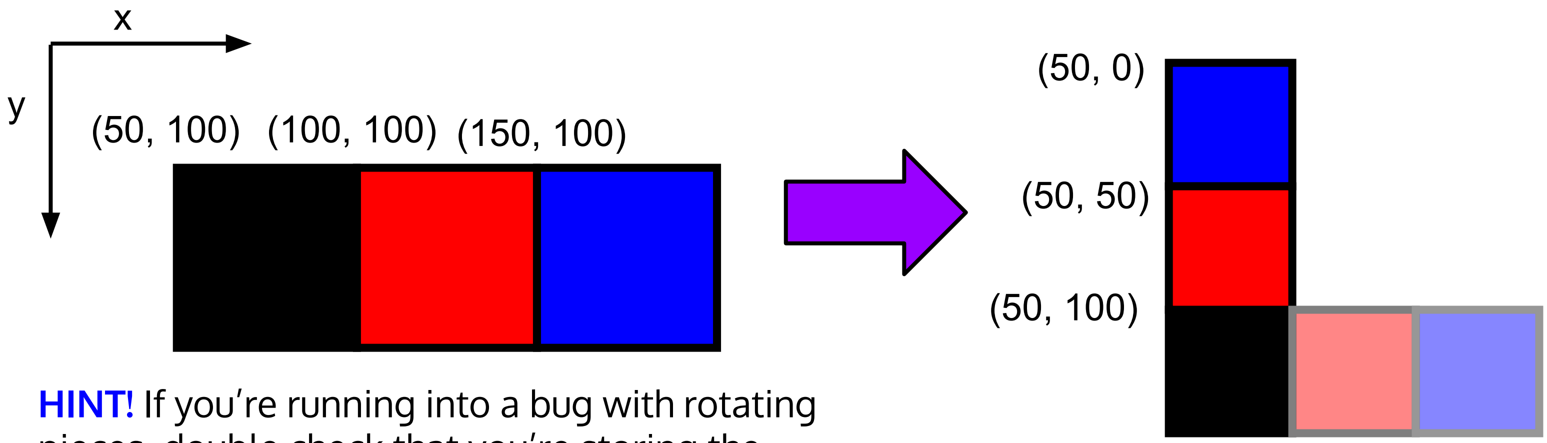
New X = CoRx - CoRy + OLy
= 50 - 100 + 100 = **50**

New Y = CoRx + CoRy - OLx
= 50 + 100 - 150 = **0**

Graphical Rotation Example (3/3)

Blue Square
Center of Rotation = (50,100)
Old Location = (150, 100)
New Location = (50, 0)

Red Square
Center of Rotation = (50,100)
Old Location = (100, 100)
New Location = (50, 50)



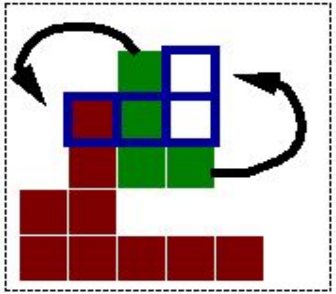
HINT! If you're running into a bug with rotating pieces, double check that you're storing the center of rotation at the start of the method - it should stay the same throughout the method.

What if it can’t move?

- Can’t move or rotate off the end of the board.
    - How does a square know if the position is off the end?
        - Check that new position is inside of the border!

What if it can’t move?

- Can’t move into a position already occupied by old pieces. The green piece cannot rotate into the blue outlined squares:



- But wait! There has to be an efficient way to keep track of the fallen squares…

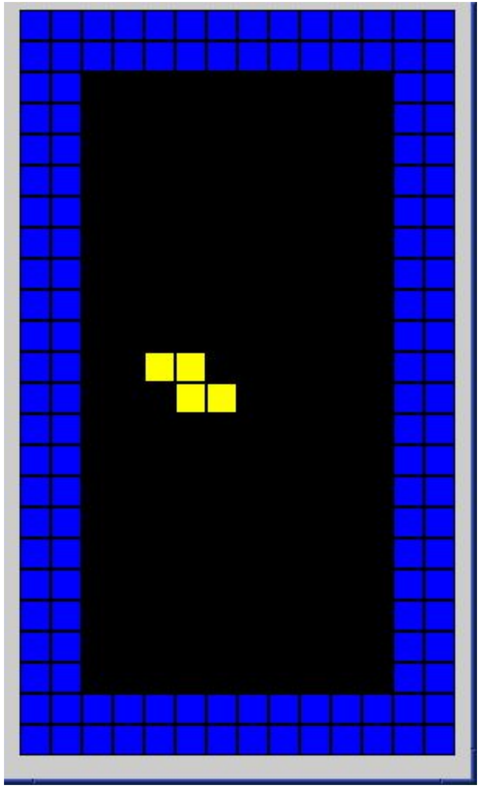
The Tetris Board

- Responsibilities:
  - Keeping track of where squares have landed.
  - The current piece needs to communicate with the board to check if its next desired move is legal.
  - Used to check for full rows.
  - Used to check for end of game.
- How do we do this?
- Read on, grasshopper.

What is the Board?

- As in previous designs, your `Game` class will use a `Pane`, to display the nodes that represent the different elements of the game.

- In Tetris, your `Game` class should also **contain** a data structure that represents the board, and that can be used for the game's functionality.



Handling Fallen Squares

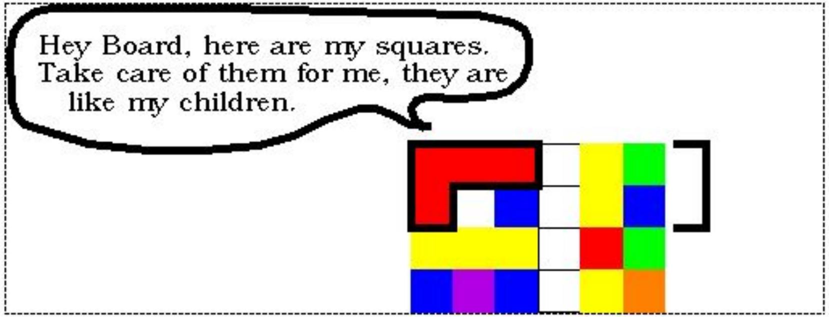
- So what data structure can we use for the Tetris `_board` to store and organize squares?
  - **2-D Array**

    (Remember, an array is a *collection of elements*, not a separate class!)
- We can use it to find/store a square located at (x, y)
  - *Everything* done to the array must be reflected on the screen!
  - *Everything* changed on the screen (except the currently moving piece) should be reflected in the array!

- **Note**: Remember that array index (row, col) is not the same as pixel (y, x).
  - Unless your squares are 1 pixel by 1 pixel
    - We would not suggest this. We would highly, *highly*, not suggest this.

Detecting and Deleting Lines

- When a piece lands, **Game** adds the piece's squares to the **_board** array.
- Then, **Game** uses **_board** to check if any horizontal lines were filled.
- Filled lines should be removed and pieces above it should be moved downward.

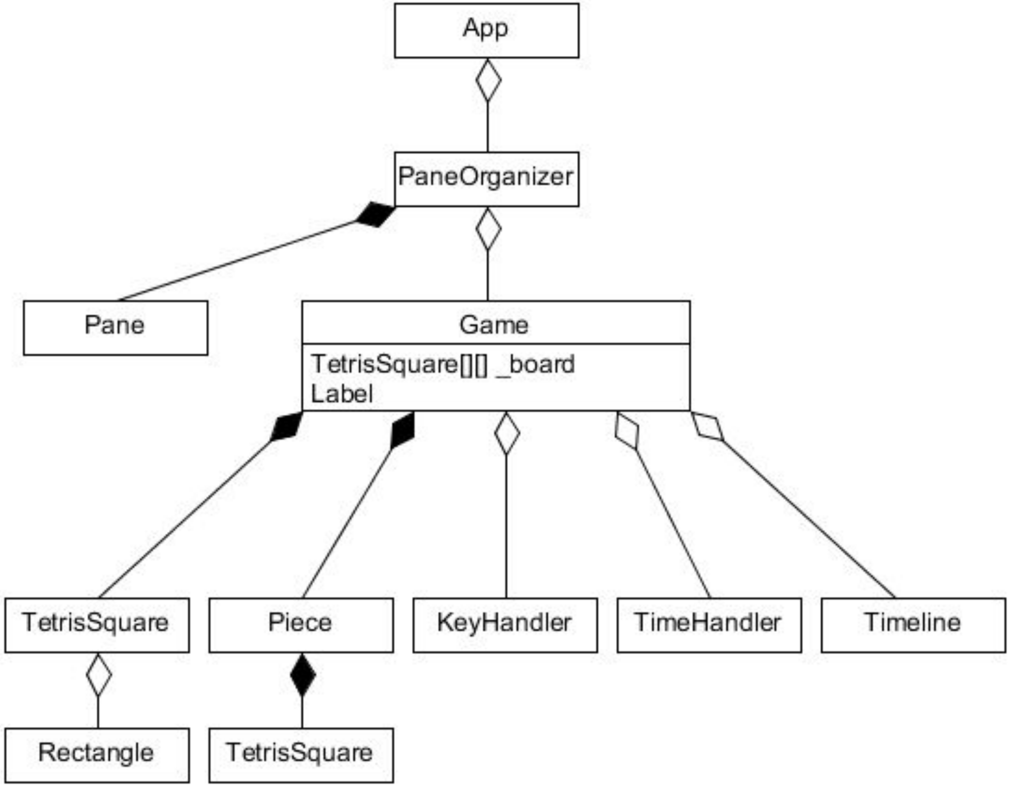

Hey Board, here are my squares. Take care of them for me, they are like my children.

End of Game

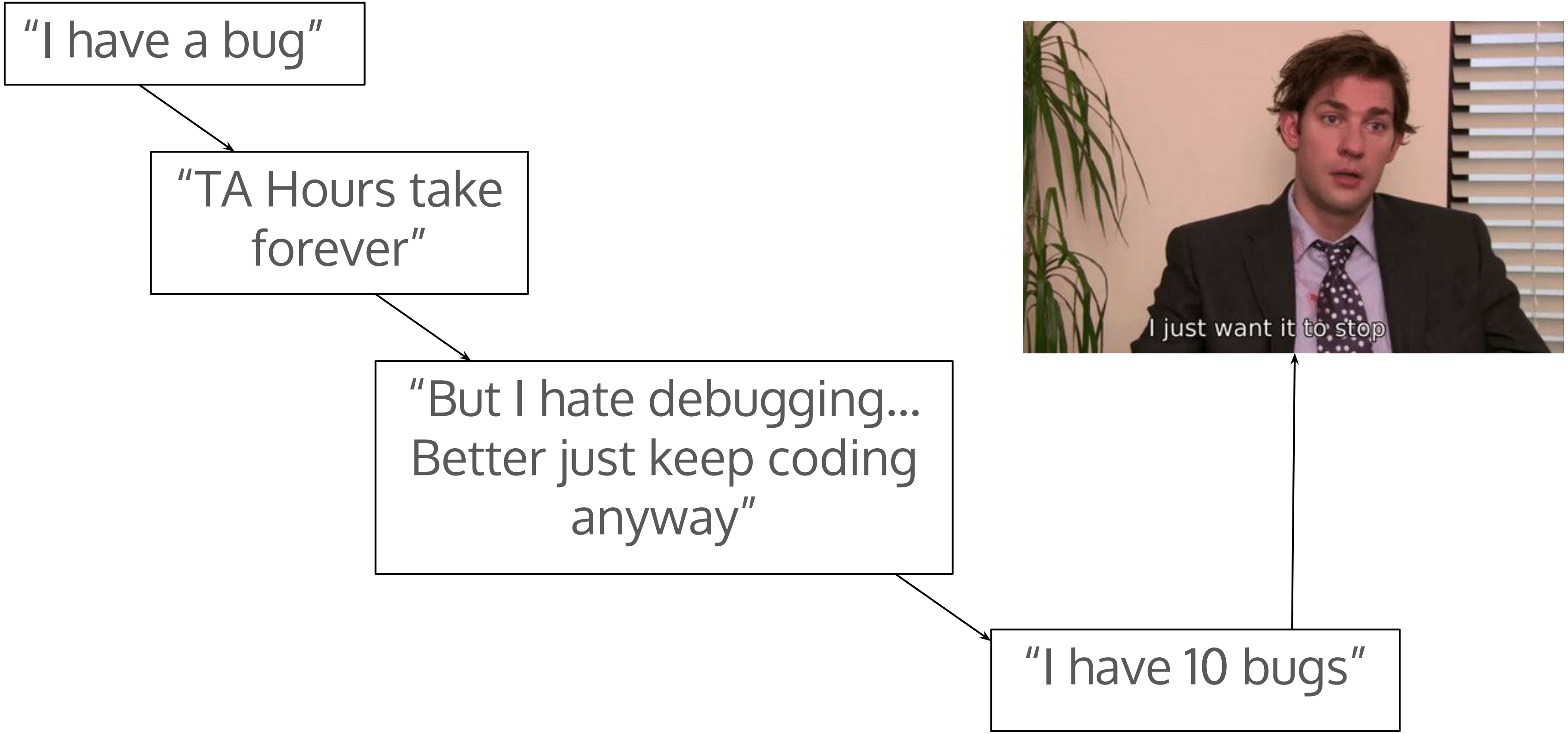
- **After `Game` handles horizontal lines, it checks for end of `Game`.**
  - `for` every location in the top row of the `_board`, `if` any location is occupied by a square, then the game is over.
  - Or, if new `Piece` cannot fall because there is no room for it to do so.
- **If condition is fulfilled, Game is over:**
  - Stop any new Tetris pieces from falling
  - Nicely tell user that game has ended (think, `Label`s!)
- **If the game is *not* over:**
  - Make a new random piece and continue

Design

- Here is one possible containment diagram for Tetris
    - Remember! There are several acceptable designs – if you want to discuss a different design, come see a TA. ***Justify design choices in your header comments!***
- **Note:** Some of the GUI portions of the containment diagram are omitted, since you have done similar layouts before.



One Note on Coding...



"I have a bug"

"TA Hours take forever"

"But I hate debugging... Better just keep coding anyway"

"I have 10 bugs"

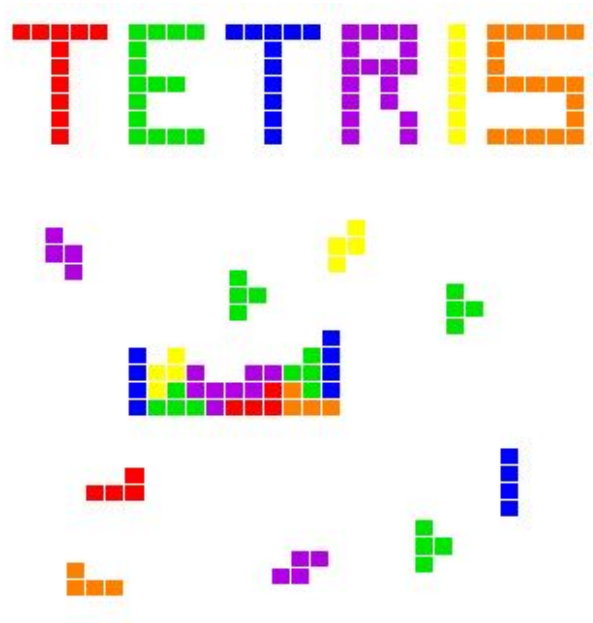
Code incrementally and debug!

- **Get one part of your code *compiling, running and producing the right visual results* before you move on to the next step**
  - This will save you time, effort, and frustration
  - The TAs will love you like no one else
- **As you code:**
  - “What have I already done?”, “What am I doing next?”
- **When you run into a bug:**
  - “When was the last time it was working the way I wanted it to?”
  - “What have I changed since then?”
- **Eclipse & Debugging**
  - Print lines!!
  - Check highlighted code

Coding Modularly

Roadmap for Tetris:

1. Get your board to show up
2. Make a piece show up
3. Make different pieces show up
4. Make pieces move/rotate
5. Make pieces not move into the border
6. Implement line clearing
7. Implement game over
8. **Do not implement extra credit until all requirements are satisfied**



GOOD LUCK!