# Programming Assignment 5 ( 100 Points )
# **Due: 11:59pm Thursday, October 27th**

## README ( 10 points )

You are required to provide a text file named **README,** NOT Readme.txt, README.pdf, or README.doc, with your assignment in your pa5 directory. There should be no file extension after the file name "**README**". Your README should include the following sections:

**Program Description ( 4 points ) :** Provide a *high level* description of your program. Describe what your programs do and how you can interact with them. Make these explanations such that your grandmother or uncle or someone you know who has no programming experience can understand what this program does and how to use it.

Write your README as if it were intended for a 5 year old. **Do not assume your reader is a computer science major.**

**Short Response ( 6 points ) :** Answer the following questions:

Vim related questions (keyboard commands only – no mouse actions, ASCII characters only):

1. In vim, how do you move the cursor to the end of a line with a single command? To the beginning of a line with a single command?
2. How do you highlight/select a line in vim?

Java related questions:

3. What does the keyword static mean in regards to variables?
4. What does the keyword static mean in regards to methods? Provide an example use of a static method.
5. What is overriding? Give an example from a previous/current assignment.

Unix related questions:

6. Using the *cut* command, how do you extract out only the columns 5 through 13 in a file named foo?

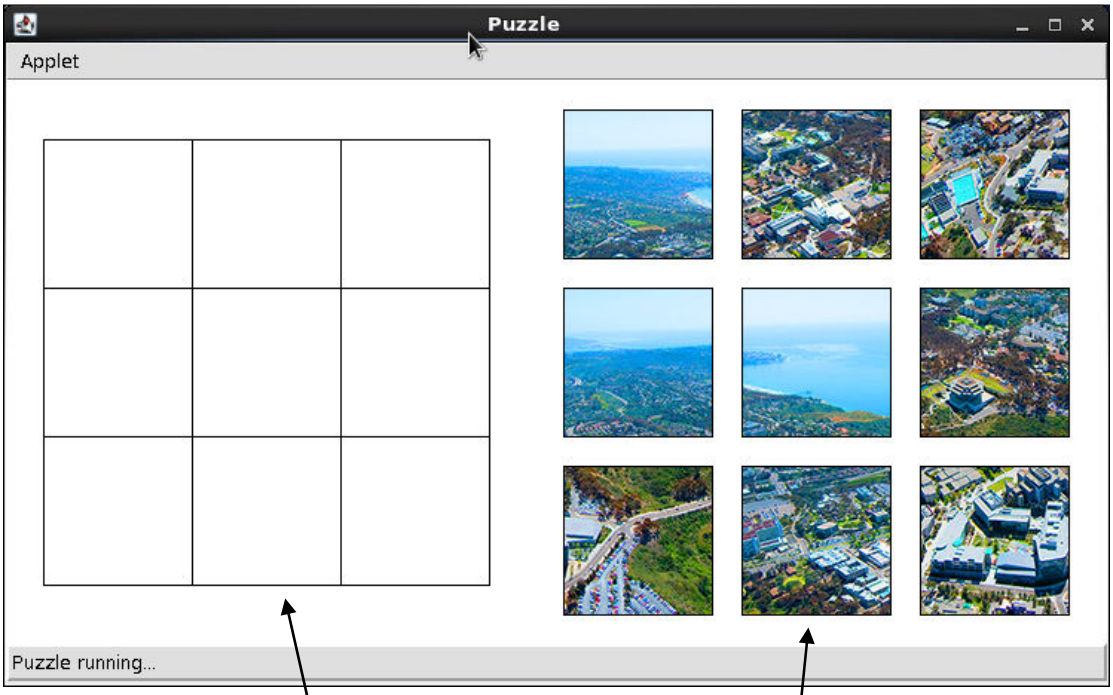
## STYLE ( 20 points )

Please see previous programming assignments. A full file header for your README is not needed, but you still need to put your name and cs11f login at the top. However, all other **style guidelines apply to all the files** ( e.g. over 80 characters apply to READMEs as well. )

You will be specifically graded on commenting, file headers, class and method headers, meaningful variable names, sufficient use of blank lines, not using more than 80 characters on a line, perfect indentation, no magic numbers/hard-coded numbers other than zero and +/-1, and use of accessor/mutator methods to access private fields (getters and setters) where specified.

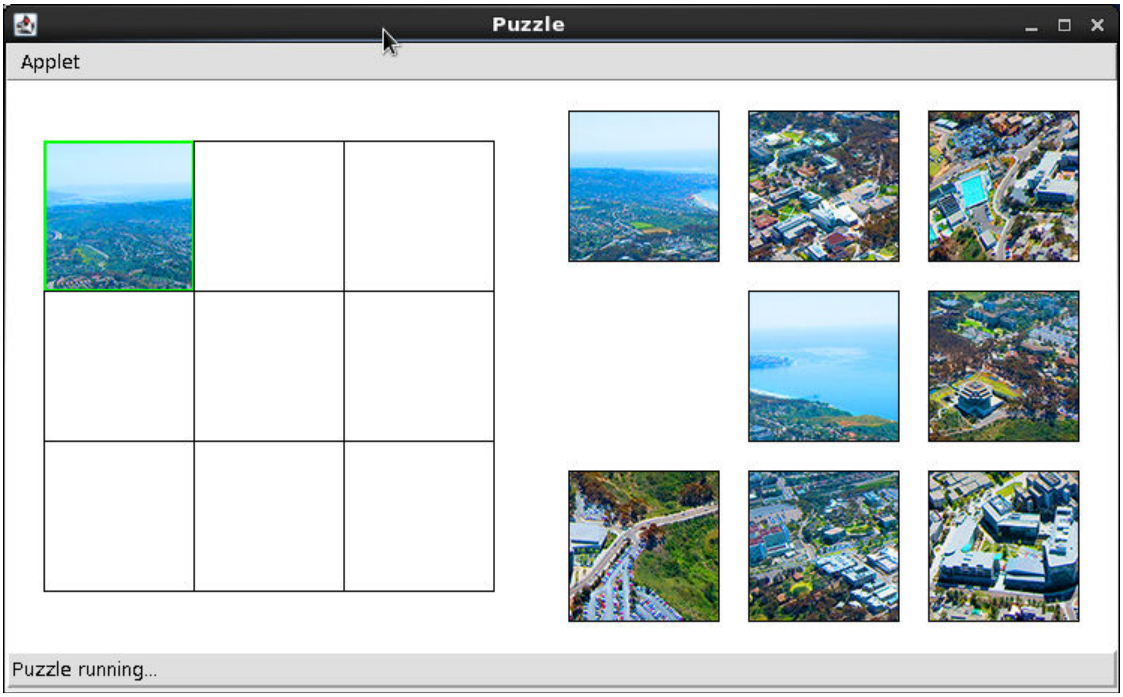
## CORRECTNESS ( 70 points )

You are asked to create an interactive 3 x 3 puzzle game for pre-schoolers. The player has to drag the puzzle pieces on the right to the correct location on the left to form a complete image. This assignment does NOT involve ActiveObjects or require a paint() method. The correctness of this assignment is based on meeting the required functionalities listed below.




Part 0: Overview

We have a 3 x 3 puzzle, which gives us 9 puzzle pieces that are movable on the right. They can be dragged anywhere on the canvas, but not off the canvas. On the left, we have a 3 x 3 board. After a puzzle piece is dragged onto its correct location, the puzzle piece is "locked" into the board and is no longer movable.



The reason "locked" is in quotes is because the puzzle piece is actually not locked. We created this illusion by having two different types of Pieces here: PuzzlePiece and BoardPiece. When a PuzzlePiece (movable piece on the right) is dragged onto a BoardPiece (fixed piece on the left), your program will do some verification to see if they match. If so, it's going to hide the PuzzlePiece and reveal the BoardPiece with the same image as the PuzzlePiece. So it may seem like the PuzzlePiece is locked in place, but it's actually a different BoardPiece that's displayed on the canvas. Now that we know the basic idea/hack behind this game, how do we implement it?

## Part 1: Useful Interfaces

Take advantage of interfaces! We have two different kinds of Piece types here, but they share a lot in common. We'll implement the Piece interface later in PuzzlePiece and BoardPiece classes.

1. The **Highlightable** interface is used to draw borders and highlights around each piece:

```
import java.awt.Color;

public interface Highlightable {
  public abstract void showHighlight( Color color );
  public abstract void hideHighlight();
}
```

2. The **Hideable** interface is used to hide and show pieces:

```
public interface Hideable {
  public abstract void show();
  public abstract void hide();
}
```

3. The **Piece** interface, which extends the two interfaces above and includes a few other methods:

```
import objectdraw.*;

public interface Piece extends Highlightable, Hideable {
  public abstract boolean contains(Location point);
  public abstract boolean equals(Object o);
  public abstract Location getCenter();
  public abstract int getId();
  public abstract void move(double dx, double dy);
}
```

## Part 2: Piece Classes

You will create 2 Piece classes: **class BoardPiece** and **class PuzzlePiece**.

Each of these classes will implement the Piece interface. For example:

```
public class BoardPiece implements Piece { ... }
```

Each Piece class will hold the specifics about that piece:

- the piece image (a VisibleImage created from passing an Image object to its constructor)
- ID (integer indexed from 0 to 8 in order of left to right and top to bottom, each ID represents a piece with a distinct image)
- center (a Location object that represents the center point of this piece)
- borders/highlights (PuzzlePiece (the movable pieces on the right) has one black FramedRect object right on top of the edge of the image and wraps around it as the border. BoardPiece (the fixed pieces on the left) has two FramedRect objects, one right on top of the edge of the image, and one that's inset by 1px,

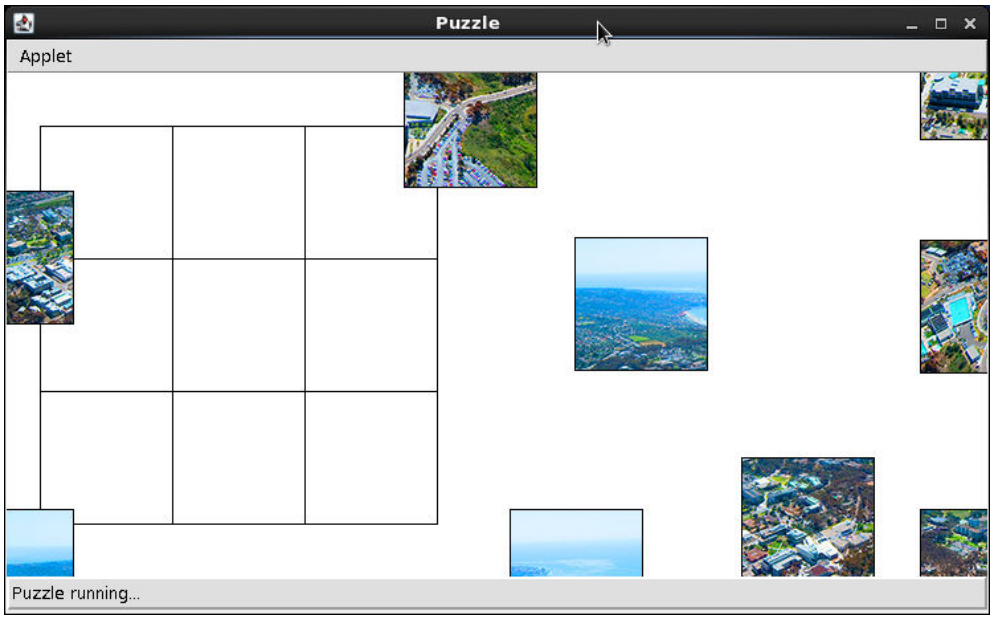
so it creates a thicker outline. When the game first starts, BoardPiece should only display the outer highlight as black.)

Thus, the constructor for BoardPiece and PuzzlePiece should be identical. For example:

```
public BoardPiece(Image img, int id, Location loc, DrawingCanvas canvas) { ... }
```

Each Piece class will also implement the methods from the Piece interface.

- **showHighlight()** and **hideHighlight()** can access and change the visibility of the FramedRect objects that highlight each piece.
- **show()** in PuzzlePiece should show the image and the black FramedRect highlight. **show()** in BoardPiece should just show the image. We will call **showHighlight()** when we want to display the green highlight to indicate the puzzle piece is in the correct spot on the board (more on this later).
- **hide()** in PuzzlePiece should hide the image and the highlight. **hide()** in BoardPiece does nothing unless you plan to implement the extra credit.
- **contains()** in PuzzlePiece determines whether some point is contained in the VisibleImage associated with that piece (Hint: use VisibleImage's contains() method). **contains()** method in BoardPiece requires a little more logic. If you are giving this puzzle to someone, you don't want him/her to solve this so quickly by just dragging the PuzzlePieces close enough to the BoardPieces so they get locked in automatically. If you are the player, you also don't want to be forced to drag the PuzzlePiece so your cursor/PuzzlePiece falls perfectly within the BoardPiece. So what we are going to do here is add a different check: instead of passing the point where the cursor is, we are going to pass the center of the PuzzlePiece we are dragging to contains(). If the center of the PuzzlePiece is within a 50 by 50 pixels square in the center of the BoardPiece, then we can say "the point is contained" in the BoardPiece. What we are really checking here is whether the center of the PuzzlePiece is contained in the 50 x 50px square in the BoardPiece.
- **equals()** checks if two Pieces are equal (for example, does the PuzzlePiece match with the BoardPiece in the board grid?). This is done by checking the unique ID (0 - 8) we assigned to each piece.
- **getId()** returns the unique ID of the Piece.
- **getCenter()** returns the center of the Piece.
- **move()** in BoardPiece should do nothing since BoardPieces are fixed. You would have to add some implementation if you decide to do extra credit. **move()** in PuzzlePiece is in charge of moving the PuzzlePiece around. There's one tricky thing about this method… you need to make sure that the PuzzlePieces cannot be dragged off the canvas. More specifically, the center of PuzzlePiece should not be able to move off the canvas.



Part 3: class Puzzle

Now we've set up the building blocks for this game, we can start implementing the controller Puzzle class. Class Puzzle should extend WindowController and include main() that starts a new instance of Acme.MainFrame. The canvas size should be **735 by 380 pixels**.

Part 3a: Get Images

Nine 100px x 100px puzzle piece images are available for you in **~/../public/PA5-images/**. They are numbered from p0 to p8 (e.g. p0.jpg, p1.jpg, …).

In order to load these images into the program, use the getImage() method in class Puzzle (the main GUI controller that implements WindowController). For example:

```
getImage("p0.jpg");
```

We need to load 9 images into an Image array (array of Image objects). Instead of hardcoding 9 lines of getImage(), we can do this in a loop using some string concatenation. getImage() will return an Image object. You should pass each Image object as an actual argument to each the of Board/PuzzlePiece's constructor, and inside that Board/PuzzlePiece's constructor, make a VisibleImage object based on the Image object.

Part 3b: Layout

The main Puzzle class will also layout the BoardPieces and the PuzzlePieces as seen in the earlier screenshots. Again, instead of hardcoding 18 different locations, we are going to use two arrays to store Locations, one for BoardPieces and one for PuzzlePieces. The good news is that the constants and the formulas of calculating these Locations have been worked out for you:

```
private static final int PIECES_PER_COL = 3; // can be changed to expand the puzzle
private static final int PIECES_PER_ROW = 3; // can be changed to expand the puzzle
private static final int PUZZLE_SPACING = 20; // num of px between PuzzlePieces
private static final int PUZZLE_OFFSET = 355; // offset from left side of canvas
private static final int BOARD_MARGIN_X = 25; // left margin of the board
private static final int BOARD_MARGIN_Y = 40; // top margin of the board
private static final int SIDE_LENGTH = 100; // side length of each Piece
```

BoardPiece with index i is located at

```
x = BOARD_MARGIN_X + SIDE_LENGTH * (i % PIECES_PER_COL)
```

and

```
y = BOARD_MARGIN_Y + SIDE_LENGTH * (i / PIECES_PER_ROW).
```

PuzzlePiece with index i is located at

```
x = PUZZLE_OFFSET + PUZZLE_SPACING * (i % PIECES_PER_COL + 1)
    + SIDE_LENGTH * (i % PIECES_PER_COL)
```

and

```
y = PUZZLE_SPACING * (i / PIECES_PER_ROW + 1) + SIDE_LENGTH * (i / PIECES_PER_ROW).
```

These x and y coordinates represent the upper left corners of the VisibleImage in each Piece. You can check the math if you are curious what's up with all these mods and divides.

Before you place all the Pieces onto the canvas, add a Text object as the winning message "YOU WON!" at x = 355 and y = the Y value of the fourth BoardPiece. It should have font size of 55 in bold and green. Hide the winning message for now, and we will come back to it later (the order of creation matters here because it affects the drawing arrangement).

Now let's put the Pieces onto the canvas. Placing the BoardPieces is easy. We already have the Image array and the Location array for BoardPieces. We just need another array of the interface type Piece to keep track of the BoardPieces. We will go through the Image array and the Location array and create BoardPieces in order. Each BoardPiece ID should match with its index in the Piece array.

For PuzzlePieces though, we want to pick one of the nine images randomly. We can do so by implementing a method called getRandomPuzzlePiece() that takes in Location and DrawingCanvas. This method will use a RandomIntGenerator that randomly generates an integer between 0 and 8. We will first check if this randomly generated index represents an image that's already used. If so, we want to get another random index until we find an unused one, because every PuzzlePiece should have a unique image. Once we've found an unused index, we can create a new PuzzlePiece. We are going to return this Piece we randomly created and store a reference to it in a Piece array. We'll do this 9 times to place all the PuzzlePieces. Note that the ID in PuzzlePiece is NOT associated with its random array index but with the unique piece image.

To avoid using magic numbers like 8, you can create a MAX_NUM_OF_PIECES constant. In our case, MAX_NUM_OF_PIECES is 9. When we need to use this as an index, we can do MAX_NUM_OF_PIECES – 1, which takes care of the offset from zero-based indexing.

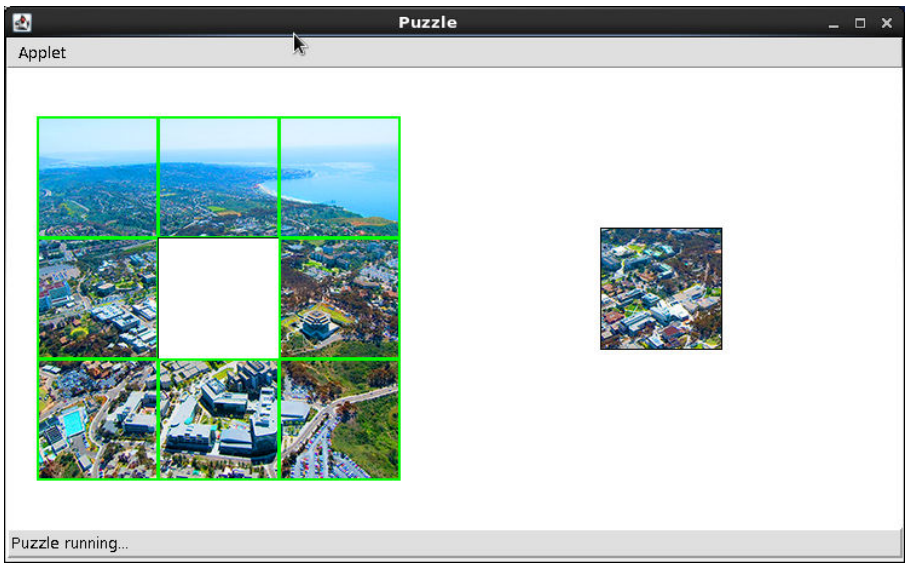
Hooray! Now you should be able to see all the BoardPieces and PuzzlePieces on your canvas. But we are not quite there yet. Your GUI controller (class Puzzle) needs to handle all the mouse events and the logic of verifying the Pieces. Take a break before you proceed. :)

## Part 3c: Dragging PuzzlePiece

By PA5, you should be pretty familiar with onMousePress(), onMouseDrag(), and onMouseRelease(). The logic is still the same. You want to keep a reference to lastPoint and find the difference between lastPoint and current point. You do need to do one additional thing here, and that is keep track of which PuzzlePiece is being grabbed. You can do so by looping through the Piece array holding all the PuzzlePieces and find which PuzzlePiece contains the point.

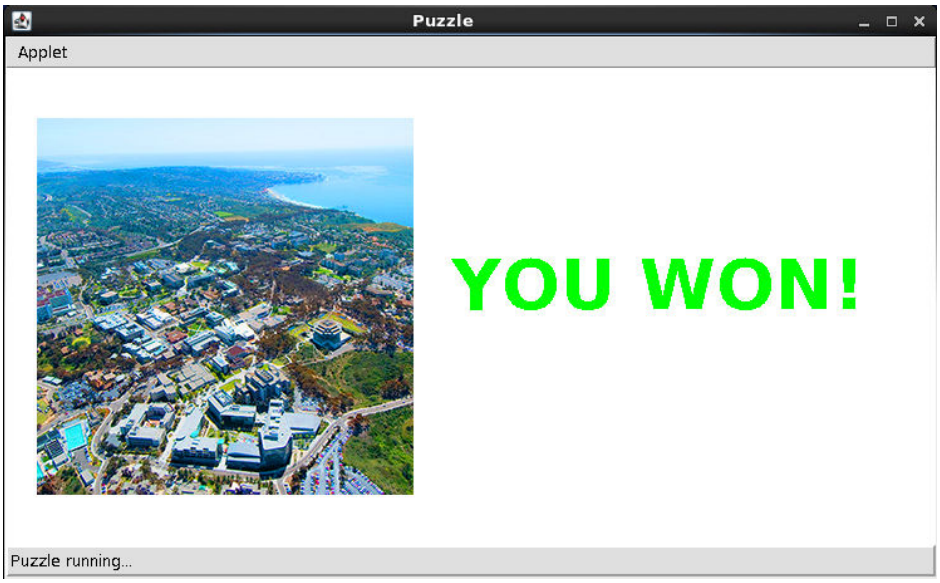
## Part 3d: Verifying PuzzlePiece

Hmmm… Where should you be checking if the puzzle piece has been placed (not while still dragging) in the correct spot on the board to the left? There's only one place, and I'll let you answer it (HINT: it is NOT in onMouseDrag()). But I'll walk you through how to verify a puzzle piece has been placed correctly. Remember many words ago, we said if the center of the PuzzlePiece is within a 50 by 50 pixels square in the center of the BoardPiece, we can say the PuzzlePiece is "contained" in the BoardPiece (this check is already done in contains()). Here, the methods you've implemented in BoardPiece and PuzzlePiece come in handy. You can use getCenter() to get the center of the grabbed puzzle piece and call contains() on every BoardPiece in a loop. If we were able to find a BoardPiece that contains the center of the grabbed puzzle piece AND they are equal (matching IDs), then we can go ahead and "lock" this Piece. Again, the "locking" is done by hiding the grabbed puzzle piece and showing the BoardPiece as described in the overview. Don't forget to show green highlights around the BoardPiece to let the pre-schoolers know that they've placed the Piece correctly!



Part 3e: Verifying Victory

Almost there… I can hear the sound of victory! How can we check if the puzzle is finished? One easy way to do this is having yet another array. This array will contain booleans that represent whether a Piece has been "locked". Since the Pieces have unique IDs from 0 to 8, we can just use those as indices and mark them true as they get "locked". The game is done after every element in this boolean array becomes true. At the moment of victory, you should hide the green highlights around each BoardPiece and display a seamless image composed of 9 pieces. You should also show the winning message.




And yaaaaaay, that's the end of this assignment.

## Running

To compile your Puzzle game, type this into the terminal:

```
> javac -cp ./Acme.jar:./objectdraw.jar:. Puzzle.java
```

To run your Puzzle game, type this into the terminal:

```
> java -cp ./Acme.jar:./objectdraw.jar:. Puzzle
```

You must have all the files in the pa5 directory including Acme.jar and objectdraw.jar.

Do not have any other .java source files in you pa5 directory other than the ones required for this assignment.

## Turnin

To turnin your code, navigate to your home directory and run the following command:

```
> cse11turnin pa5
```

You may turn in your programming assignment as many times as you like. The last submission you turn in before the deadline is the one that we will collect.

## Verify

To verify a previously turned in assignment,

```
> cse11verify pa5
```

If you are unsure your program has been turned in, use the verify command. We will not take any late files you forgot to turn in. Verify will help you check which files you have successfully submitted. **It is your responsibility to make sure you properly turned in your assignment.**

## Files to be Collected

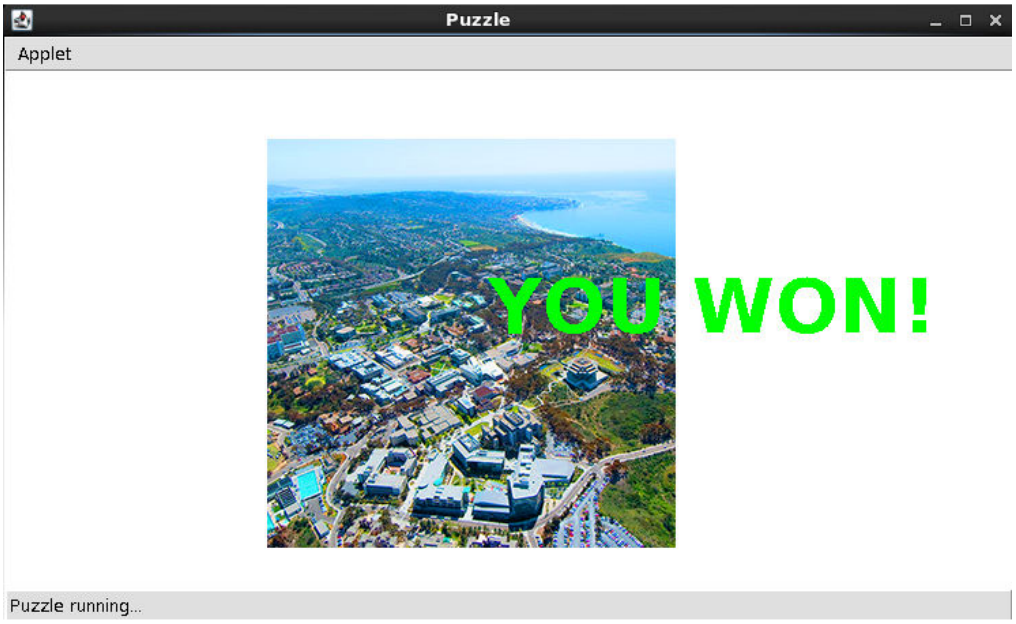
| Source Files | Images | Misc. |
|---|---|---|
| BoardPiece.java | p0.jpg | Acme.jar |
| Hideable.java | p1.jpg | README |
| Highlightable.java | p2.jpg | objectdraw.jar |
| Piece.java | p3.jpg | |
| Puzzle.java | p4.jpg | |
| PuzzlePiece.java | p5.jpg | |
| | p6.jpg | |
| | p7.jpg | |
| | p8.jpg | |

## Extra Credit ( 5 points )

**Note: There are NO separate extra credit files for this assignment. Implement the extra credit in the same files as the regular assignment. Specify in your README Program Description which parts of the extra credit you have implemented.**

1. Dragging after Winning [2.5pts total]

After the user has finished the game, the 9 BoardPieces should look like a complete image of an aerial view of UC San Diego. As a part of extra credit, enable dragging this "image" around the canvas (really, you are dragging these 9 pieces together). You will have to change the move() method inside BoardPiece. Don't worry about dragging the image off canvas (you can always resize the canvas and get it back).




2. Clicking after Winning [2.5pts total]

The pre-schoolers just can't stop playing your puzzle again and again, but their teacher is not always there to restart the game for them. To make it easier for them, add a feature so that clicking anywhere on the complete image of the puzzle will restart the game. You'd want to add an onMouseClick() and reinitialize many things (Hint: You don't need to reinitialize the Image array or the Text object). Clicking on the white space should do nothing.

**NO LATE ASSIGNMENTS ACCEPTED!**

# START EARLY!

...and **HAVE FUN!**