Programming Assignment 5 (PA5) - TileGame

Due Date: Wednesday, October 31 @ 11:59 pm

<table>
<thead>
<tr>
<th>Assignment Overview</th>
<th>Grading</th>
<th>Gathering Starter Files</th>
<th>Background Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>TileGame</td>
<td>README File</td>
<td>Extra Credit</td>
<td>Turnin Summary</td>
</tr>
</tbody>
</table>

**Assignment Overview**

In this assignment, you will make a fun and interactive tile puzzle game to demonstrate your knowledge of interfaces. There is only 1 program for this assignment. Please refer to the demo video posted on piazza (as well as the screenshots in the writeup) to verify that your program is working correctly.

**Grading**

- **README: 10 points** - See README Requirements [here](http://cseweb.ucsd.edu/~ricko/CSE11READMEGuidelines.pdf) and questions below
- **Style: 20 points** - See Style Requirements [here](http://cseweb.ucsd.edu/~ricko/CSE11StyleGuidelines.pdf)
- **Correctness: 70 points**
- **Extra Credit: 5 points** - View Extra Credit section for more information.

**NOTE:** If what you turn in does not compile, you will receive 0 points for this assignment.

**Gathering Starter Files**

You will need to create a new directory named `pa5` and go into that directory. The $ represents your command prompt. What you type in is in **bold**.

```bash
$ mkdir ~/pa5
$ cd ~/pa5
```
Copy the starter files from the public directory.

```bash
$ cp ~/../public/objectdraw.jar .
$ cp ~/../public/Acme.jar .
$ cp ~/../public/PA5StarterCode/PA5Constants.java .
$ cp ~/../public/PA5StarterCode/Timer.java .
```

### Starter files provided:

- objectdraw.jar
- Acme.jar
- PA5Constants.java
- Timer.java

---

## Background Info -- Random Number Generators & Command Line Arguments

### Java Random Number Generators

In this assignment you will be generating random numbers using a random number generator. Computers (at least the ones we're dealing with) don't have any “random” components in them, so they can't produce actually random numbers. Darn. This means we go for the next best thing--pseudorandom!

Pseudorandom number generators work by generating a predictable sequence of numbers. To give the effect of randomness, you as the programmer need to “seed” the random number generator to essentially set the starting place of where in the sequence the generator will start returning values to you. The “seed” is simply an integer. A corollary of this is that the same seed will always generate the same sequence of “random” numbers--use this to your advantage! This is super helpful for testing!

### Command Line Arguments

#### Entering Command Line Arguments:

So now that we know we need to seed the random number generator, we want to make this a user-provided value so that we don't need to change our code and recompile every time we want to use a different seed. Also, if we hard-coded the value to use as the seed, it would produce the same game every time which isn't very fun. To make things easy, we'll have the random number seed be a required command line argument. This means when we run the program, rather than typing:

```bash
java -cp ./objectdraw.jar:./Acme.jar:. TileGame
```

we will add the seed for the random number generator as an argument at the end of the run command:

```bash
java -cp ./objectdraw.jar:./Acme.jar:. TileGame 420
```

So in this case, 420 would be used as the seed for the random number generator.

#### Processing Command Line Arguments:

Now that we know how to enter command line arguments as the user, we need to learn how to process those command line arguments in our code. In Java, the command line arguments are automatically passed as an array of Strings to main():

```java
public static void main(String[] args) {...}
```

For example (command line arguments are bold):

```bash
java -cp ./objectdraw.jar:./Acme.jar:. TileGame 420
```
Note that any whitespace between the command line arguments is trimmed and **not** stored in the `args` array.

To access these values within `main()`, we can just treat it as a regular array (because that's what it is). In the case of example 1, we can access the String "420" by writing `args[0]` because "420" is stored at index 0 of the `args` array; In the case of example 2, we can access the String "cse" by writing `args[2]` because "cse" is stored at index 2 of the `args` array; and so on.

**Putting it all Together**

So now that we know all about random number generator seeds and command line arguments, we can put it all together. The goal is to create a random number generator where the seed is is entered by the user as a command line argument.

First let's go take a look at the javadocs for `java.util.Random` (go ahead, click on the link, it won’t bite. I'll wait). So if we look at the “Constructor Summary” we can see there are two constructors: one that takes no parameters, and one that takes a seed as a parameter. Hopefully by now you’ve guessed that we want to use the constructor that takes in a seed as a parameter.

But wait! `args` stores the seed as a String, but this constructor takes in a long! Ah, fear not, because you are smart, and you did PA2 where you learned how to convert a String to an int. For our purposes, we can just pass the seed to the constructor as an int (it will automatically be converted to a long). So once we have the seed as an int, we can create the random number generator as follows (note that in the example we are hardcoding the number for the seed, but in your code you will be getting the value for the seed by converting the command line argument from a String to an int):

```java
int seed = 420;
Random randomGenerator = new Random(seed);
```

And it’s as simple as that! Now that we know how to create a random number generator, let's figure out how to use it. Go ahead and look back at the javadocs linked above. Yes, actually go do it. Now scroll down to “Method Summary”. We will be using the random number generator to generate random integers between 0 and some max value, so every time we want to get a random number, we will use the following method. For example, if we wanted a random number between 0 and 24, we would write:
Some important things to take note of (seriously please read this):

- Do NOT, I repeat, do not create a new Random object every time you want to get another random number. Creating a new Random object every time you want to get another random number is like buying a new coffee pot every time you want to make another cup of coffee. There is nothing wrong with the old coffee pot; use it.
- In the example above, the random number generator is created as a local variable. In your program, you will need to create the Random object as an instance variable in TileGame, that way it can be used in other methods in the class.

Details on where all this fits into your code are provided in the next section below.

TileGame

You are asked to create an interactive 3 x 3 tile game for preschoolers. The player has to swap colored tiles in the grid on the left so that they match the arrangement of the colored tiles in the grid on the right. This assignment does NOT involve ActiveObjects or require a paint() method.

Part 0: Overview

There are two 3 x 3 grids made up of colored tiles. The grid on the right is the solution grid (made up of SolutionTiles). The SolutionTiles should not respond to any mouse events. The grid on the left is the active grid (made up of ActiveTiles). The player plays the game by interacting with the ActiveTiles (ActiveTiles respond to mouse events).

When the player clicks on a tile in the active grid, the tile is selected and highlighted in green. Clicking on the selected tile again deselects it. Once there are two tiles selected, the user can swap the 2 selected tiles by long-clicking (click longer than PA5Constants.SWAP_THRESHOLD) on either one of them. There can be up to 2 tiles that are selected and highlighted at any one time, and only after deselecting one of them or swapping them can the user select another tile. If any tiles in the grid on the left match their counterparts in the solution grid on the right (and therefore are in their correct locations), the matched tiles in the left and right grids are highlighted in magenta and the tile in the left is no longer swappable or selectable.
When program starts, it is possible that some pieces are already in the correct place (highlighted in Magenta)

When you click on an ActiveTile, it gets selected and highlighted as green

When two ActiveTiles are selected, you can long-click on either of them to swap them

The two tiles got swapped and notice that we have one more correct tile now! (The middle tile in the first row)

**Part 1: Useful Interfaces**

Take advantage of interfaces (by this we mean you **have to** use them)! Each interface defines a set of behaviors the tiles should have, and should be implemented accordingly.

1. The **Colorable** interface is used to make each Tile a specific color:

   ```java
   import java.awt.Color;

   public interface Colorable {
       public abstract Color getColor();
       public abstract void setColor(Color color);
   }
   ```

2. The **Highlightable** interface is used to draw borders and highlights around each tile:

   ```java
   import java.awt.Color;

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

3. The **Selectable** interface is used to select tiles:
   ```java
   import objectdraw.*;
   public interface Selectable {
       public abstract boolean contains(Location point);
   }
   ```

4. The **Swappable** interface is used to swap tiles:
   ```java
   public interface Swappable {
       public abstract void swap(Colorable colorable);
       public abstract boolean isSwappable();
       public abstract void setSwappable(boolean swappable);
   }
   ```

At this point, you’re probably wondering, “Do I really need to have method and class headers for all these interfaces and methods?” I will save you the trouble of piazza and tell you: Yes you need method and class headers for all the interfaces. However, you do not need method headers for inherited methods inside a class that implements an interface. Note that you still need method headers for any non-inherited methods. If you ask this question on piazza, you will be directed back to the writeup. If you go back to piazza and ask this question again, you will have learned about infinite loops.

**Part 2: Tile Classes**

You will create 2 tile classes: `ActiveTile` and `SolutionTile`.

- **SolutionTile** will implement the `Colorable` and `Highlightable` interfaces:
  ```java
  public class SolutionTile implements Colorable, Highlightable { ... }
  ```

- **ActiveTile** will implement the `Colorable`, `Highlightable`, `Swappable` and `Selectable` interfaces:
  ```java
  public class ActiveTile implements Colorable, Highlightable,
      Swappable, Selectable { ... }
  ```

Each tile class will hold the specifics about that tile:

- The tile, a `FilledRect` of a certain color.
- The highlight/border/frame, a `FilledRect` of the highlight color that sits behind the tile to create a border effect.

Thus, the constructor for `ActiveTile` and `SolutionTile` should be identical. For example:

```java
public SolutionTile(Color color, Location loc, DrawingCanvas canvas) { ... }
```
Each tile class will also implement the methods from the Colorable and Highlightable interfaces:
- `getColor()` returns the color of the tile.
- `setColor()` sets the color of the tile.
- `showHighlight()` displays the highlight with the color passed in by changing the color of the FilledRect behind the tile to the provided color.
- `hideHighlight()` resets the highlight to the default highlight color.
- `isHighlighted()` returns whether or not the tile is highlighted.

ActiveTile also needs some additional functionality because it also implements the Selectable and Swappable interfaces:
- `contains()` returns whether or not the provided point is inside the tile (includes the highlight/border).
- `swap()` swaps the tile’s color with that of another Colorable.
- `isSwappable()` returns whether or not the tile is swappable (hint: if the tile is in its correct location, it should no longer be swappable).
- `setSwappable()` sets the tile to be locked or free to be swapped.

Note that all of the constants and Strings for this program are in PA5Constants.java.

**Part 3: Class TileGame**

Now that we’ve set up the building blocks for this game, we can start implementing the GUI controller: TileGame. Class TileGame should extend WindowController and include a main() method. Since we have a command line argument this time (the seed for the random number generator—if this makes no sense, go read the Background Info section up above), main() will need to perform the following:

- First make sure that exactly 1 command line argument was entered. If there is not exactly 1 command line argument, print the usage message and exit with code 1 (look back at the PA3 writeup if you don’t know what this means).
- If there was exactly 1 command line argument, try converting the argument to an integer. If this fails, print the usage message and exit with code 1.
- Now that we have the seed for the random number generator as an integer, create a new instance of Acme.MainFrame as usual, but this time pass in the int seed as a parameter to the TileGame constructor. The canvas size should be **735 by 380 pixels** (check out the constants in PA5Constants.java).

Inside the TileGame constructor, you’ll want to create the random number generator, using the seed provided as a command line argument.

If an error is encountered while parsing the command line argument, your output must look like the following:

```
[cs1ffxxx@ieng6]:pa5$ java -cp ./objectdraw.jar:.Acme.jar:. TileGame
Usage: TileGame seed
   seed: must be a valid integer
[cs1ffxxx@ieng6]:pa5$
```
Usage: TileGame seed
       seed: must be a valid integer

Usage: TileGame seed
       seed: must be a valid integer

Part 3a: Layout
The main TileGame class will lay out the ActiveTile grid and the SolutionTile grid. Instead of hardcoding 18
different locations, we are going to use two arrays to store Locations, one for ActiveTile and one for
SolutionTile. The good news is that the constants and the formulas of calculating these Locations have been
worked out for you in PA5Constants.java.

ActiveTile with index i is located at
   x = PA5Constants.BOARD_MARGIN_X +
       PA5Constants.FRAME_SIDE * (i % PA5Constants.TILES_PER_COL);
   y = PA5Constants.BOARD_MARGIN_Y +
       PA5Constants.FRAME_SIDE * (i / PA5Constants.TILES_PER_ROW);

SolutionTile with index i is located at
   x = PA5Constants.PUZZLE_OFFSET + PA5Constants.BOARD_MARGIN_X +
       PA5Constants.FRAME_SIDE * (i % PA5Constants.TILES_PER_COL);
   y = PA5Constants.BOARD_MARGIN_Y +
       PA5Constants.FRAME_SIDE * (i / PA5Constants.TILES_PER_ROW);

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

Part 3b: Randomly generate colors and place tiles on canvas
Now, let's place the tiles on the canvas. In order to make the colors of the tiles random, but still solvable, there
is a set order of operations needed to generate the colors for the SolutionTile as well as the ActiveTile. Note
that since this random ordering is based on the random seed given by the user, it is very important that you
follow the below steps exactly, otherwise your program will not match with the same seed.

First, you should define a constant array of the possible tile colors:
   private static final Color[] COLORS = { PA5Constants.TILE_COLOR_1,
                                             PA5Constants.TILE_COLOR_2, PA5Constants.TILE_COLOR_3,
                                             PA5Constants.TILE_COLOR_4, PA5Constants.TILE_COLOR_5 };
Create the SolutionTiles:
For each of the locations determined above, generate a random index into the COLORS array and place a SolutionTile with that color at that location on the canvas. Generate the tiles in the order left to right, top to bottom (this order is very important). As you create each SolutionTile, save it into an array for later use.

Create the ActiveTiles:
When creating the grid of ActiveTiles, you must generate the same number of tiles with each color as the SolutionTile grid because we need to be able to finish the game. That is, if the solution grid has 4 blue tiles and 5 red tiles, the active grid also needs to have 4 blue tiles and 5 red tiles.

To do this, create an array of booleans the size of the number of tiles with all of the boolean values starting off as false. Each boolean indicates if the SolutionTile with the same index in the array of SolutionTiles has a corresponding tile in the ActiveTile grid yet. For each ActiveTile location (again, going from top left to bottom right), generate a random index into the boolean array, check if the boolean at that index is false, meaning the SolutionTile with that index does not have a corresponding tile in the ActiveTile grid yet. If the boolean is true, generate another random index until you find one where the boolean is false.

Now that you have an index for a SolutionTile that you have not already used, create the ActiveTile that corresponds to the SolutionTile (make sure to save the ActiveTile in the array of ActiveTiles) and set the boolean at that index to true. Continue doing this until you have placed all the tiles on the canvas.

This way, when we are generating the ActiveTile grid, we are shuffling the order of colors of tiles in the SolutionTile grid, but we keep the number of tiles with each color the same to guarantee that there is a solution.

If you have correctly generated the tile colors, the color ordering should be the same as in the demo video, given that the same seed is used.

After you generate all the tiles, check the colors of the ActiveTiles against the SolutionTiles to highlight matched ones in magenta (PA5Constants.MATCH_COLOR) on both sides.

Last, add a Text object with the winning message (PA5Constants.WIN_MSG) at x position PA5Constants.PUZZLE_OFFSET and y position of the ActiveTile at index 3. There are constants in PA5Constants.java for the font size and color of the win message. Also note that you need to set the font to bold. Hide the message so it is not initially seen.

Part 3c: Selecting pieces
The player should be able to select and unselect tiles by clicking on them. If the mouse clicked on an ActiveTile that is not highlighted, highlight it in Color.GREEN to indicate that it is selected. If the tile is already selected, deselect it (there’s an interface method for this). There should be at most 2 tiles selected at any one time, which means if two tiles are currently selected, you can’t select a new tile without first deselecting one of the selected ones first (this is something the user has to do, not something you should write in your code).

Part 3d: Swapping pieces
The player should be able to swap two selected tiles by long-clicking on one of them. You are not actually swapping the ActiveTile object. Instead, swap their colors using methods defined in the Colorable interface. Once tiles are swapped, they are no longer selected.
**Part 3e: Verifying pieces**
When tiles are swapped, check if either of the tiles are now in the correct position (a tile is in the correct position when its color matches the color of the same tile in the solution grid). Once a tile is in the correct position, use one of the interface methods to mark that it is no longer swappable (to lock it in place) and then set the highlight colors of the ActiveTile and its corresponding SolutionTile to magenta (PA5Constants.MATCH_COLOR).

**Part 3f: Verifying victory**
If all of the ActiveTiles are matched (in their correct positions, matching the tiles in the solution grid), the player won the game! Once this happens, display the win text. You'll need to check for victory every time that tiles could match.

---

**README File**

1. In vim, in command mode, how do you move forward one word in a line of code? Move back one word? (Not using arrow keys and only using a single key)
2. How do you jump to a certain line number in your code with a single command in vim? For example, how do you jump directly to line 20? (Without arrow keys)
3. What are the only things that can be declared in a Java interface? Be specific with full access modifiers and keywords.
4. What run time structure holds the local variables and formal parameters with each method invocation?
5. Why are professional engineers expected to act with integrity?

---

**Extra Credit**

- **[5 Points]** Make a button to reset the game to its original state after you win.

**Getting Started:**
Make a copy of your TileGame file to do the extra credit in.

```
$ cd ~/pa5
$ cp TileGame.java EC_TileGame.java
```

**Important:** Your original TileGame.java file must remain unchanged. You need both the regular and the EC version of this file for turnin.
EC Requirements:
Implement a reset button. At any point in the game, you can click on the reset button to reset the game back to how it was started -- so that you can play more rounds of the same game! It is up to you how you want to implement the reset button or what you want it to look like. Be creative!

Reset button

Turnin Summary
See the turnin instructions here. Your file names must match the below *exactly*.

Due Date: Wednesday night, October 31 @ 11:59 pm

Files required for the Turn-in:
ActiveTile.java    Acme.jar
Colorable.java    objectdraw.jar
Highlightable.java PA5Constants.java
Selectable.java    Timer.java
SolutionTile.java
Swappable.java
TileGame.java

Extra Credit Files:
EC_TileGame.java

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.
NO EXCUSES!
NO EXTENSIONS!
NO EXCEPTIONS!
NO LATE ASSIGNMENTS ACCEPTED!
DO NOT EMAIL US YOUR ASSIGNMENT!

Start Early, Finish Early, and Have Fun!