P02 Water Fountain

Overview

This assignment involves developing a graphical implementation of a particle system. Particle systems have long been used in computer graphics to render amorphous objects like: fire, clouds, and water. The idea behind this technique is to use several small images or shapes that together give the appearance of a large amorphous object that does not necessarily have a clearly defined surface.[1] The Graphical User Interface (GUI) of this assignment will be written using the java processing graphic library. Fig.1 shows an example of what this program might look like when it is done.

Figure 1: P02 Water Fountain Display Window

[1] Although not required for this assignment, interested students are welcome to read this early paper about the use of particle systems in early movie effects including Star Trek II: The Wrath of Khan

Learning Objectives

The goals of this assignment include:

- Practice how to create and use objects,
- Develop the basis for creating an interactive graphical application, and developing your own test methods.
- Give you experience working with callback methods to define how your program responds to mouse or key based input.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P02 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 02/06/2022**.
25 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
20 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.

Assignment Requirements and Notes

- Pair programming is **NOT ALLOWED** for this assignment. You MUST complete and submit your p02 INDIVIDUALLY.
- The ONLY import statements that you may include in this assignment are:

```
import java.util.Random;
import java.io.File;
import processing.core.PImage;
```

- This may be your first experience with developing a graphic application using the java processing library. Make sure to read carefully through the specification provided in this write-up. This assignment requires clear understanding of its instructions. Read TWICE

the instructions and do not hesitate to ask for clarification on piazza if you find any ambiguity.

- The automated grading tests in gradescope are **NOT using the full processing library** when grading your code. They only TWO methods that you are allowed to use from the `PImage` class in your submisison on gradescope are:
  - PImage `loadImage(String)`, and
  - void `image(PImage,int,int)`.

  If you are using any other fields, methods, or classes from the `PImage` class, this will cause problems for the automated grading tests.
- You MUST NOT add any additional fields either instance or static to your program, and any public methods either static or instance to your program, other than those defined in this write-up.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private static methods to help implement the different public static methods defined in this program, if needed.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- **NONE** of the callback methods `(setup(), draw(), mousePressed(), and keyPressed()` defined later in this write-up should be called explicitly in this program.
- There is NO tester class to be implemented for this assignment. Your tester methods will be implemented directly in your `Fountain` class and called as first statement in your `Fountain.setup()` method.
- All the methods in this programming assignment MUST be static. We are focusing on procedural programming through p04.
- You can submit your work in progress multiple times on gradescope. Your submission may include methods not implemented or with partial implementation or with a default return statement. But avoid submitting a code which does not compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- Your submitted source file must include at the top a complete file header.
- All implemented methods including the main method MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.
- All your classes MUST have a javadoc-style class header.
- You MUST adhere to the Academic Conduct Expectations and Advice

1 GETTING STARTED

1. To get started, let's first create a new Java11 project within Eclipse. You can name this project whatever you like, but "P02 Water Fountain" is a descriptive choice.

2. Then, create a new class named `Fountain` with a public static void main(String[] args) method stub. This class represents the main class in your program. This process is described near the end of the Eclipse installation instructions here.

3. DO NOT include a package statement at the top of your `Fountain` class (leave it in the default package). The Fountain.java source file that this class is defined within will be the only file that you submit for grading through gradescope.com.

1.1 Download p2Fountain.jar file and add it to your project build path

Download this P2Fountain.jar file and copy it into the project folder that you just created. Then, right-click on this file in the "Package Explorer" within Eclipse, choose "Build Path" and then "Add to Build Path" from the menu. Note: If the .jar file is not immediately visible within Eclipse's Package Explorer, try right-clicking on your project folder and selecting "Refresh". This jar file that you are using makes use of the processing library, which you will get more direct experience with in future assignments.

(**Note that** for Chrome users on MAC, Chrome may block the the jar file and incorrectly reports it as a malicious file. To be able to copy the downloaded jar file, Go to "chrome://downloads/" and click on "Show in folder" to open the folder where your jar file is located.)

If the "Build Path" entry is missing when you right click on the jar file in the "Package Explorer", follow the next set of instructions to add the jar file to the build path:

1. Right-click on the project and choose "Properties".

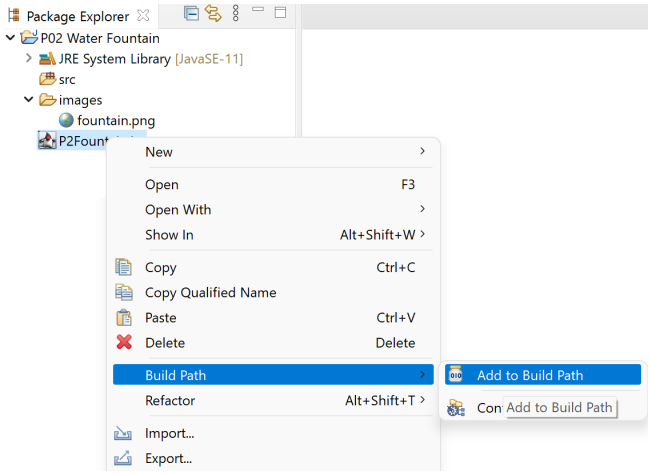
2. Click on the "Java Build Path" option in the left side menu.

3. From the Java Build Path window, click on the "Libraries" Tab.

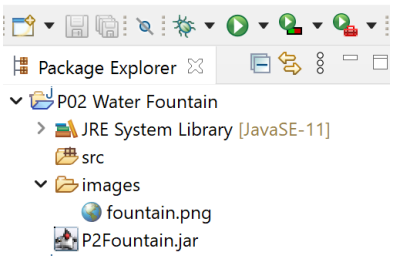
4. You can add the "P2Fountain.jar" file located in your project folder by clicking "Add JARs..." from the right side menu.

5. Click on "Apply" button.

This operation is illustrated in Fig.2 (a).



(a) Adding P2Fountain.jar to build path



(b) P02 package explorer

Figure 2: Fish Tank Application - Getting Started

1.2 Check your project setup

Now, to test that the P2Fountain.jar file library was appropriately added to the build path of your project, try running your program with the following method being called from `main()` method.

```
Utility.runApplication(); // starts the application
```

If everything is working correctly, you should see a blank window that appears with the title, "P02 Water Fountain" as depicted in Fig.3 along with an error message in the console which we'll resolve in the next steps: `ERROR: Could not find method named setup that can take arguments [] in class Fountain`.

Please consult piazza or one of the consultants, if you have any problems with this setup before proceeding. Note that the processing library provided within the jar file (P2Fountain.jar) works with java 11 ONLY. If you work with another version of java, you must switch to java 11 for this assignment.

Note that the runApplication() method from the provided Utility class, provided in the P2Fountain jar file, creates the main window for the application, and then repeatedly updates its appearance and checks for user input. It also checks if specific **callback methods** have been defined in the `Fountain` class. Callback methods specify additional computation that should happen when the program begins, the user pressed a key or a mouse button, and every time the display window is repeatedly redrawn to the screen.

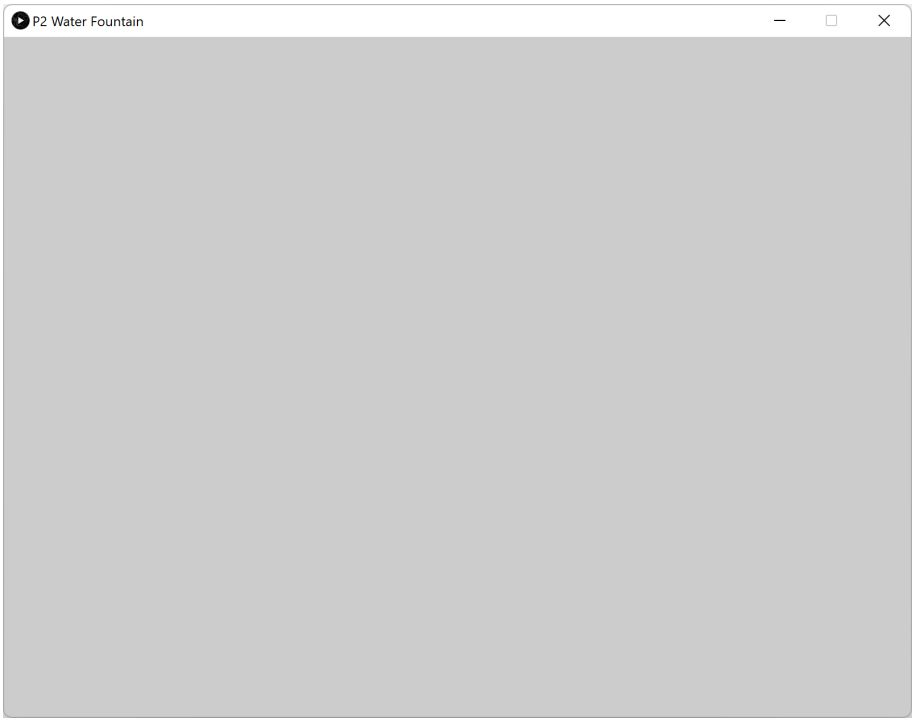


Figure 3: P02 Water Fountain - Blank Screen Window

1.3 Download the image of the fountain

Create a folder called images within your project folder in Eclipse: this can be done by right clicking your project in the "Project Explorer" and selecting $New > Folder$. Then, download this fountain image and copy it into your new "images" folder by following the links, then right-clicking on the images, and selecting "Save Image as...". If you don't see them in Eclipse immediately, try right-clicking on your project folder and selecting "Refresh".

The organization of your p02 project through eclipse's package explorer is illustrated by Fig. 2 (b).

2 Utility Framework and Overview of the class Droplet

All of the methods within the provided P2Fountain.jar file are documented in these javadocs. Note that the `Utility.runApplication()` creates the display window, sets its dimension, and checks for callback methods. A callback method specifies additional computation that should happen in response to specific events.

2.1 Overview of Callback Methods

In this assignment, we are going to implement the following callback methods.

- `setup()` method: This method is automatically called by Utility.runApplication() when the program begins. It creates and initializes the different data fields defined in your program, and configures the different graphical settings of your application, such as loading the background image, setting the dimensions of the display window, etc.
- `draw()` method: This method continuously executes the lines of code contained inside its block until the program is stopped. It continuously draws the application display window and updates its content with respect to any change or any event which affects its appearance.
- `mousePressed()`: The block of code defined in this method is automatically executed each time the mouse bottom is pressed.
- `keyPressed()`: The block of code defined in this method is automatically executed each time a key is pressed.

Note that NONE of the above callback methods should be called explicitly in your program. They are automatically called by every processing programs in response to specific events as described above. You can read through the documentation of the Utility class and have an idea about the above callback methods and the other methods which can be used to draw images and circles to the screen. You can also read through some of the other methods that are available through this Utility class. Note that many of these methods take floats rather double values as arguments. Any time that you need to express a float constant in Java, you should use a lower case f to distinguish its type: 1.3 is a double, but 1.3f is a float.

2.2 Overview of the class Droplet

The Droplet class represents the data type for droplet objects that will be created and used in our `Fountain` application. The javadoc documentation for this class is provided here. Make sure to read the description of the constructor and the different methods implemented in the Droplet class carefully. You do not need to implement this class or any of its declared methods. The class Droplet is entirely provided for you in the P2Fountain.jar file. Its implementation details are hidden for you as users of that class. All the important information required to use its public methods appropriately are provided in these javadocs.

3 Visualizing the Water Fountain Display Window

3.1 Define the setup and draw callback methods

At the end of the last step we saw an error related to the fact that our Fountain class was missing a method called setup(). Let's solve that problem.

- Create a **public static** method with named `setup` that **takes no input arguments** and **has no return value**.
- Next, run your program. This should lead to a slightly different error message being displayed: `"ERROR: Could not find method named draw that can take arguments [] in class Fountain."`
- We can now fix this error in a similar way, by creating a new **public static** method named `draw`, which also **takes no input** arguments and **has no return** value.
- If you look at the JavaDoc documentation for the Utility class, see how the `runApplication` method is making use of the setup and draw methods that we have just created. Recall that these methods should not be called directly from your code. They should only be called by the Utility class, as a result of your calling Utility.runApplication().
- To convince yourself that the `setup()` method is being called once when your program starts up, try adding a print statement to the `setup()` method, then run your program. Note that the output of your System.out.println() print statement will be displayed in the console and not on the graphic display window. After this test, you can remove the print statement.
- To convince yourself that the `draw()` method is being repeatedly called by the Utility class as a result of your Utility.runApplication() call, try adding a print statement to the `draw()` method. Note that the text message will be repeatedly printed to the console until you terminate the program. After this test, you can remove the print statement before proceeding.

3.2 Set the background color and draw some circles

- We're going to be utilizing a lot of randomly generated values throughout this assignment, so let's create a `private static Random` field named `randGen` within the Fountain class. If you are not familiar with this class, you can read about the `nextInt()` and `nextFloat()` methods within the JavaDocs here. The `randGen` data field must be defined outside of any method. The top of the class body is a good placement where to declare data fields.
- Next, initialize the `randGen` field to a new Random object within your `Fountain.setup()` method. You do not need to use a specific seed when instantiating this random number generator, although doing so may be helpful debugging your code
- Then, within the `setup()` method, pass a randomly generated int value to `Utility.background()`.
- Now, every time you run this program, the background will be cleared to a different random color.
- Next, try drawing some circles using the `Utility.circle()` method defined in the Utility class. You can draw circles with different sizes at different position of the screen.

  - Note that the position `(0, 0)` refers to the upper left corner of the display window.
  - The position `(Utility.width(), Utility.height())` refers to the bottom right corner of the display window.
  - The position `(Utility.width()/2, Utility.height()/2)` refers to the center of the display window.

- To change the color of the circles that are drawn, you can call `Utility.fill()` method before calling the `Utility.circle()` method. You can provide a randomly generated int to the `Utility.fill()` method call.

- Although randomly chosen colors can be fun, let's specify some particular colors that we'll continue to use going forward. We are going to use the `Utility.color()` method to specify the mix of red, green, and blue light that make up the colors that we'll want to use here.

  - We are going to use `Utility.color(253,245,230)` for the background color.
  - We'll later draw a circle on that background with the color `Utility.color(23,141,235)`.

- To do so, add the `Utility.background(Utility.color(253,245,230))` and `Utility.fill(Utility.color(23,141,235)` method calls to the `draw()` method. Then, delete any line of code from the `setup()` method which draws a circle.

- Before proceeding, check that your `setup()` method contains only one line of code which initializes the `randGen` data field to a new Random object.

3.3 Draw a fountain image at the top of the background

To draw an image of a water fountain at the top of the display window, follow the next instructions.

1. Create a `private static PImage` field named `fountainImage` within the class `Fountain`. This data field will represent the image of the fountain. If you get a compiler error, make sure to `import processing.core.PImage;` to resolve it.

2. Create a a `private static int` field named `positionX` within the class `Fountain`. This data field will represent the x-position where the image of the fountain will be drawn to the screen.

3. Create a a `private static int` field named `positionY` within the class `Fountain`. This data field will represent the y-position where the image of the fountain will be drawn to the screen.

4. Now, in the `setup()` method, initialize the `positionX` and `positionY` data fields to `Utility.width()/2` and `Utility.height()/2`, respectively. This will set the initial position of the fountain to the center of the screen.

Next, make sure that you have a folder called "images" in your "Package Explorer" and that it contains one image "fountain.png". If it is not the case, go up to "Download the image of the fountain" subsection in the Getting Started section.

In the `setup()` method, load the fountain image and store its reference into the `fountainImage` variable of type PImage by calling the method `Utility.loadImage()` as follows.

```
// load the image of the fountain
fountainImage = Utility.loadImage("images" + File.separator + "fountain.png");
```

To avoid compile without errors, check that `java.util.Random`, `java.io.File`, and `processing.core.PImage` classes are already imported at the top of your Fountain.java source file. If you run now your program, you can notice that even though the image of the fountain was loaded in the setup() method, it is not yet drawn at the center of the screen.

- To draw the fountain image to the screen, we need to call the `Utility.image()` method from the `draw()` method. The `Utility.image()` method draws a PImage object at a given position of the screen. Please read the javadocs of the Utility class for more details. You can find below an example of code that draws an image at the center of our Water Fountain display window. (Recall that we initialized the positionX and positionY to the center of the screen within the `setup()` method.)

  ```
  // Draw the fountain image to the screen at position (positionX, positionY)
  Utility.image(fountainImage, positionX, positionY);
  ```

- Notice the importance of running the above line of code within the `draw()` method after calling the `Utility.background()` method, instead of before. The `Utility.background()` clears the display window each time it is called as first line of code within the `draw()` method.

Now, if you run your program should result in a window comparable to the one shown in Fig.4.

4 Animation

If you haven't already, look through the JavaDoc documentation for the Droplet class that we'll be using next.

- Create a `private static Droplet array` field named `droplets` within your Fountain class. This array defines a perfect size array of Droplets.
- Then, within your Fountain's setup() method: initialize the `droplets` field to a new `Droplet array` containing a single reference to a single Droplet object (we'll make this array bigger, with more droplets later).

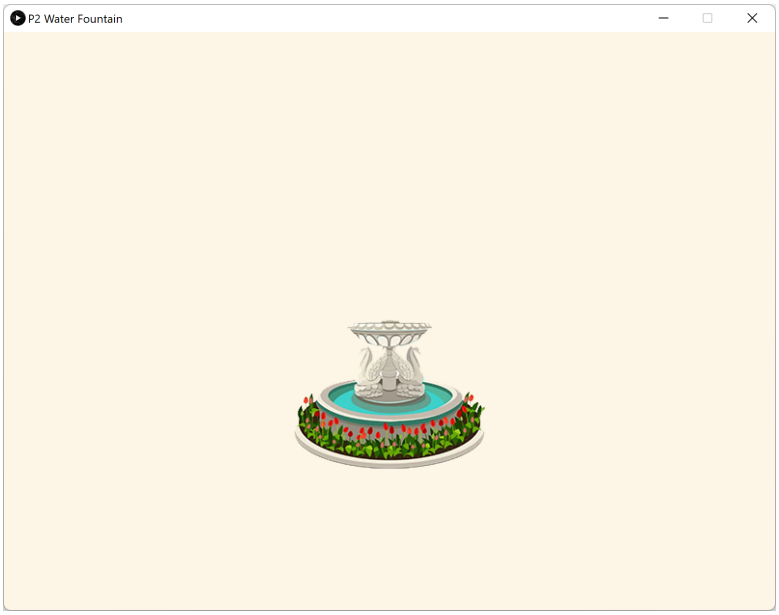


Figure 4: A fountain at the top of the background

- Within your `draw()` method, draw a circle by calling the `Utility.circle()` method. Then, modify this code to make use of the size, color, transparency, and position of your droplet object (that you stored within the droplets array). The circle should have the `Utility.color(23,141,235)` as fill color. Try to organize your code such that the method call of `Utility.fill()` always proceeds the method call of `Utility.circle()`.

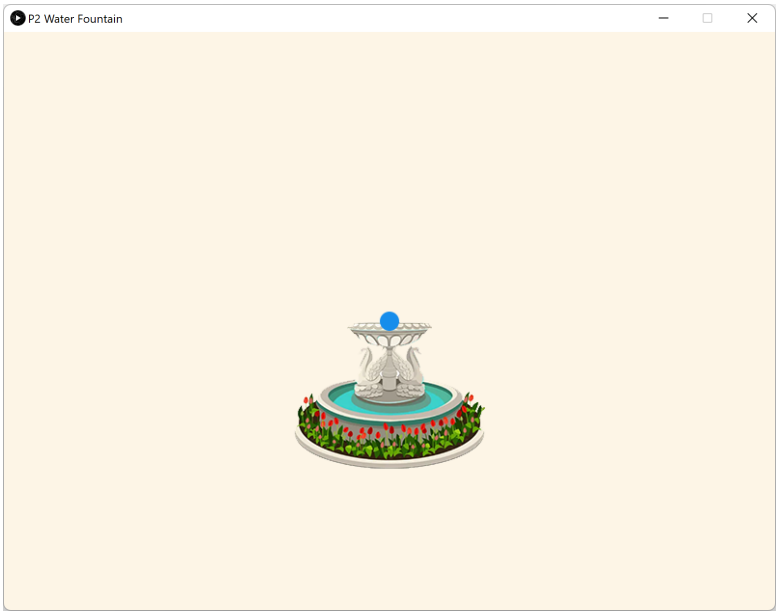


Figure 5: One droplet at the top of the water fountain

If you run now your program, you should have the output illustrated by Fig.5.

Next, we'll try gradually changing the position of our droplet over time, by doing so from within the Fountain's draw() method that is being repeatedly called. Notice that each droplet has its own `velocityX` and `velocityY` state.

- Set the x and y velocity of your droplet to **-6** within `Fountain.setup()` method.
- You'll notice that this doesn't automatically update the position of your droplet. Instead, you'll need to add code to your `Fountain.draw()` method which increments the `x position` by the `x velocity`, and increments the `y position` by the `y velocity`. This should result in positive velocities causing position values to increase, and negative velocities causing position values to decrease.

Running your program at this point should result in the droplet moving up and to the left (because both velocities are negative). If you can see more than one droplet circle at a time, try moving the `Utility.background` call from `setup()` to the beginning of your `draw()`, to fix this. Next, we'll add a gravitational effect.

- In the `draw()` method, before the lines of code moving the droplet, adjust the `y velocity` of that droplet to by adding **0.3f** to it. This should result in what looks like a ball being tossed, as this downward acceleration causes the droplet to sweep out a parabola.

5 More droplets

Let's first update the size of your `droplets` array from 1 to 800 within the `setup()` method, leaving only a non-null droplet reference at index 0.

5.1 Update a droplet

Ultimately, we'd like to have several droplets being created, updated, moved, and removed from our water fountain through our `draw()` method. To prevent this method from getting too complex, we are going to begin refactoring the code within it.

1. Create a new `private static` method named `updateDroplet` that takes an **int index as input** and has **no return** value. The index passed as input represents the index of a specific droplet within the `droplets` array. This method should do all of the **moving** (by setting positions x and y), **accelerating** (updating the y-velocity by adding 0.3f to it), and **drawing** of a droplet (by calling `Utility.fill()` and `Utility.circle()`) that we already used them in the `draw()` method, but should be able to do that with whatever droplet is specified through the provided index.

2. Besides, instead of setting the fill color to `Utility.color(23,141,235)`, we would like to use the `color` and `transparency` of the **droplet being drawn**. You should use the `Utility.fill((int color, int transparency))` to implement this update.

3. Add a for loop in your `draw()` method to call the `updateDroplet` method on each droplet stored in your `droplets` array. Watch out for NullPointerExceptions as the `droplets` array may contain null references at any of its indexes.

Your `draw()` method should now be left clearing the background, drawing the fountain image, and then looping through all the indexes of your `droplets` array, and calling `updateDroplet()` for each index that does not contain a null reference. Next, we'll create a method that attempts to add more new droplets to this array.

5.2 Create new droplets

Start by removing from the `setup()` method any line of code which creates any droplet object. The `droplets` array will contain 800 null references, only when created within the `setup()` method.

Next, create a new `private static` method called `createNewDroplets` that **takes an int** number of new droplets to create as input and **has no return** value. This method should begin stepping through indexes of the `droplets` array in ascending order. As it does so, it will be looking for null references within this array that can be changed to reference newly created droplets. This loop will continue to do this until either:

1) it has stepped through all valid indexes and there are no more to check, or

2) it has created the specified number of new droplets and has stored references to them within the droplets array.

Check that each time your Fountain's `draw()` method is called, it should call `createNewDroplets()` to attempt to create **10** new droplets. Then, it makes call to the `updateDroplet()` method from each not null reference within the array `droplets`.

Create new droplets with different initial states

Now, we'll want to vary the initial state of these droplets being created within our `createNewDroplets()` method. But first let's add some `private static fields` to our Fountain class. Initialize these fields to the values shown within your `Fountain.setup()` method.

```
private static int startColor; // blue: Utility.color(23,141,235)
private static int endColor; // lighter blue: Utility.color(23,200,255)
```

Use these fields (along with the Random field `randGen`) to randomize the initial state of each new droplet that is created within your `createNewDroplets()` method as follows.

- The x position of each droplet should begin within 3 pixels (+/-) of Fountain.positionX.
- The y position of each droplet should begin within 3 pixels (+/-) of Fountain.positionY.
- The size of each droplet should begin between 4 and 11.
- The color of each droplet should begin between Fountain.startColor and Fountain.endColor (note: the `lerpColor()` defined in Utility may be helpful for this).
- The x velocity of each droplet should begin between -1 and 1.
- The y velocity of each droplet should begin between -5 and -10.
- The age of each droplet should begin between 0 and 40.
- The transparency of each droplet should begin between 32 and 127.

For int values: both the min and max bounds specified below should be considered inclusive (for instance, a droplet can have the age of 0 or 40).

For float values: the min should be inclusive, but the max should be exclusive (for instance, the x position of a new droplet could not be Fountain.positionX+3, but could be Fountain.positionX+2.999999). For instance, setting the x-position of a droplet within 3 pixels (+/-) of Fountain.positionX can be solved as follows.

```
x-position = positionX + randGen.nextFloat() * 6 - 3
```

Now, when you run this program, you should see a flurry of droplets resembling a fountain. Once the 800 spaces within your array are used up, no more droplets can be created or seen. In the next step, we'll remove some of the old droplet references so that their positions can be reused by newly created droplets.

If you haven't already, feel free to remove that funny droplet that we were creating in `setup()`, so that all of the droplets that we see are created by our `createNewDroplets()` method.

6 Removing and moving droplets

We would like in this section to remove old droplets and replace them with new ones. You may have noticed that we set each droplet to have a randomized age between 0 and 40. You are going to **increment by one** the age of a droplet every time the `updateDroplet()` method is called on it's index from the `draw()` method. Implement this now.

6.1 Remove old droplets

Then, create a new `private static` method named `removeOldDroplets` that takes a **max age int as input**, and **has no return** value. This method should be called from the end of your `Fountain.draw()` method's definition, with a **max age of 80**.

The job of this new method is to search through the `droplets` array for references to droplets with an age that is greater than the specified max age and remove them. Be careful to null references in the `droplets` array while searching for the old droplets. To remove an old droplet, you can simply set its reference to null. You do not need to make any shift operation. Null references can be at any position of a perfect size array. Implementing this correctly should allow your water fountain to continue running forever.

6.2 Implement the mousePressed() callback method

If you try clicking on the screen, you may notice an error message related to a mousePressed method that cannot be found. Let's create this public static method now. Create a `public static` method named `mousePressed`. This method must take **no input** arguments, and has **no return** value.

You should implement this method to move the `Fountain.positionX` and `Fountain.positionY` to match the position of the mouse whenever the mouse button is pressed. To implement this method, review the documentation of the Utility class to find the methods which return the x and y positions of the mouse.

Clicking on the screen will now allow you to move the entire fountain around which new droplets are created. You can also drag it around like a sparkler.

6.3 Implement the keyPressed() callback method

And last but not least, let's create a `public static` method called `keyPressed` that takes **a single char** argument representing the key that was pressed, and has **no return** value.

Within this method, call `Utility.save("screenshot.png")` whenever the key pressed happened to be the s-key (either 's' or 'S'). This is another callback method that will be called at the appropriate time by the Utility class, and so should never be called from your Fountain class.

7 Testing and reflection

Let's now write a couple of tests to confirm the behavior of a couple methods under very specific circumstances. Here are the **TWO** test methods that you MUST implement within your `Fountain` class. Since running many of your methods require first calling `Utility.runApplication()`, you can call these test methods from the beginning of your `setup()` method BEFORE ANY OF THE STATIC FIELDS ARE INITIALIZED. This will help ensure that your tests don't

interfere with the subsequent execution of your program. Note that the methods below already have JavaDoc style comments, like the ones that you are required to add to all of your other methods by the course style guide. Each of them includes one test scenario to be implemented.

```
/**
 * This tester initializes the droplets array to hold at least three droplets.
 * Creates a single droplet at position (3,3) with velocity (-1,-2). Then checks
 * whether calling updateDroplet() on this droplet's index correctly results in
 * changing its position to (2.0, 1.3).
 *
 * @return true when no defect is found, and false otherwise
 */
private static boolean testUpdateDroplet() {
  return false; // TODO: implement this test
}

/**
 * This tester initializes the droplets array to hold at least three droplets.
 * Calls removeOldDroplets(6) on an array with three droplets (two of which have
 * ages over six and another that does not). Then checks whether the old droplets
 * were removed and the young droplet was left alone.
 *
 * @return true when no defect is found, and false otherwise
 */
private static boolean testRemoveOldDroplets() {
  return false; // TODO: implement this test
}
```

8 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only ONE file that you must submit is `Fountain.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of **9:59PM on February 10**th. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.