Mutability Lab 1

Lab 1 - Drawing with Processing

The Processing language provides a low barrier to writing code with visual output. Processing is Java under the hood, so you will see the same syntax as the previous lessons. However, the structure of the code is slightly different. We are going to use Processing to make a bouncing ball animation. Click the button below to launch Processing.

info

Open the File

You have to tell Processing the file you want to open. In the Processing window, click `File` then `Open...`. On the left towards the bottom of the list, click `workspace`. Double click on `code`, double click on `mutability`, and double click on `drawing`. Finally, open the `drawing.pde` file. This file will be used for the animation.

A Processing program is built around two methods: `setup` and `draw`. The `setup` method runs one time, and is used to set the size of the window or give global variables their value. The `draw` method is an infinite loop, which makes animations easier to create. Start by setting the size of the window to 400 pixels by 400 pixels. Unfortunately, you cannot copy/paste code into Processing.

```
void setup() {
  size(400, 400);
}

void draw() {

}
```

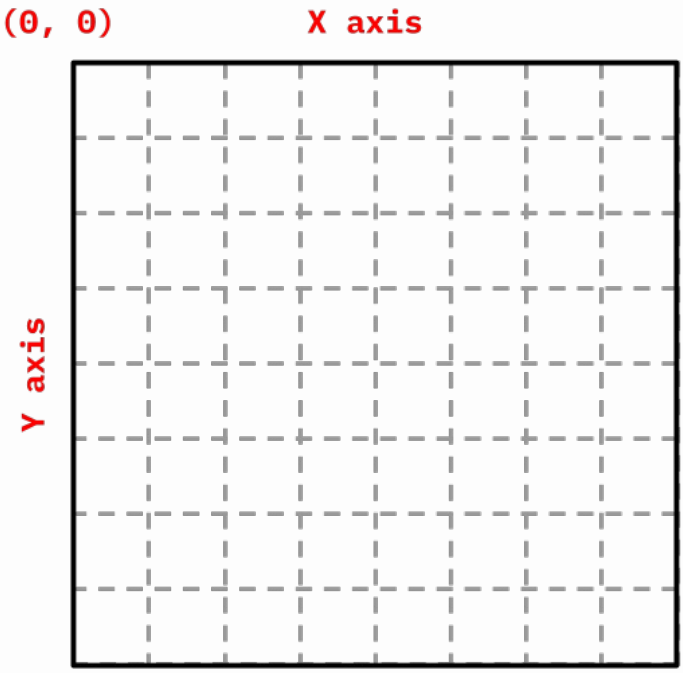
info

Try It

Click the triangle button to run you program. You should see a 400 x 400 window appear. Close the window or click the square button to stop the program. **Note**, you need to manually save your work in Processing. Click `File` and then `Save`, or press `Ctrl + S` on the keyboard to save your work.

The Window

Before drawing anything in the window, it is important to understand how the window works. It is a Cartesian plane with an x-axis and a y-axis. However, the origin point (0, 0) is not in the center of the plane. Instead, the origin point is in the top-left corner. That means the x-values increase as you move to the right. Y-values increase as you move down the window. Keep this in mind as you draw and animate shapes on the window.



The Window

Shapes and Lines

Instead of printing text like in Java, we are going to draw shapes and lines to the window. Here are how create some basic shapes and a line.

Shape	Method	Parameters
Rectangle	rect	(x, y, width, height)
Square	square	(x, y, length)
Ellipse	ellipse	(x, y, width, height)
Circle	circle	(x, y, diameter)
Line	line	(startX, startY, endX, endY)

Anything drawn to the window should be done inside the `draw` method.

```
void draw() {
  line(200, 0, 200, 400);
  line(0, 200, 400, 200);
  ellipse(100, 100, 50, 80);
  circle(300, 100, 80);
  rect(75, 260, 50, 80);
  square(260, 260, 80);
}
```

In Processing, the way circles and ellipses are drawn is different from rectangles and squares. Change the `draw` method to the code below. Both the circle and the square are drawn to the same location with the same size. However, circles and ellipse are drawn from the center of the shape, while square and rectangles are drawn from the top-left corner.

```
void draw() {
  circle(200, 200, 100);
  square(200, 200, 100);
}
```

You can find more shapes (listed under 2D Primitives) in the Processing documentation.

Color

By default, Processing gives each shape an exterior color (called the stroke) of black and an interior color (called fill) of white. You can change both of these colors, but first you need to understand that Processing used the **RGB** color scheme. Amounts of **R**ed, **G**reen, and **B**lue are mixed together to form a color. The amount is a number from 0 to 255. You can use websites to help you determine the RGB value for any color.

Use the `fill` command with an RGB value to color the interior of a shape. The `stroke` command is used to color the outline of a shape and to color lines.

```
void draw() {
  stroke(255, 195, 0 );
  line(200, 0, 200, 400);
  line(0, 200, 400, 200);

  stroke(199, 0, 57);
  fill(255, 87, 51);
  ellipse(100, 100, 50, 80);

  stroke(255, 87, 51);
  fill(199, 0, 57);
  circle(300, 100, 80);

  stroke(144, 12, 63);
  fill(88, 24, 69);
  rect(75, 260, 50, 80);

  stroke(88, 24, 69);
  fill(144, 12, 63);
  square(260, 260, 80);
}
```

challenge

Try these variations:

- Change the `strokeWeight` in the `setup` method.

```
void setup() {
  size(400, 400);
  strokeWeight(5);
}
```

▼ **What happened?**
It was hard to see the lines and stroke of each shape. `strokeWeight` determines how thick lines are drawn. Since we want all shapes and lines to have the same stroke weight, we only need to call this line of code one time, which is why it is in the `setup` method.

- Draw a rectangle at position 100, 100 with a width of 50 and a heght of 75. Choose any color you want.

▼ **Solution**

```
rect(100, 100, 50, 75);
```

- Draw a circle at position 100, 100 with a diameter of 40. Choose a different color from the rectangle.

▼ **Solution**

```
circle(100, 100, 40);
```

- Draw an ellipse at position 200, 350 with a width of 200 and a height of 50. Choose any color you want.

▼ **Solution**

```
ellipse(200, 350, 200, 50);
```

- Draw a diagonal line from the top-left to the bottom-right.

▼ **Solution**

```
line(0, 0, 400, 400);
```

▼ **The Position of the Code for Shapes and Lines is Important**

If shapes or lines are overlapping with other shapes or lines, the ones drawn last are "on top". Similarly, the `fill` method only applies to the shape or line that comes after it. The code samples below are "incorrect" because you cannot see the smaller square because it comes before the larger square. In addition, only one of the circles is pink because the `fill` methods come after the shape. Switch places for the squares and make sure all of the `fill` commands come before a shape.

```
void draw() {
  // hidden square
  fill(245, 145, 45);
  square(125, 125, 150);
  fill(45, 145, 245);
  square(50, 50, 300);

  // incorrect color placement
  circle(25, 200, 40);
  fill(244, 88, 178);
  circle(375, 200, 40);
  fill(244, 88, 97);
}
```

Mutability Lab 2

Lab 2 - Ball Class

info

Animation File

For the next three pages, you are going to use the `animation.pde` file. In the Processing window, click `File` then `Open...`. On the left towards the bottom of the list, click `workspace`. Double click on `code`, double click on `mutability`, and double click on `animation`. Finally, open the `animation.pde` file.

The purpose of this lab is to build a bouncing ball animation with objects and Processing. Before the animation can take place, the `Ball` class and its constructor need to be defined. We'll also set up the window in the `setup` method.

```
//add class definitions below this line

class Ball {
  float xPosition;
  float yPosition;
  color ballColor;
  int radius;

  Ball(float x, float y) {
    xPosition = x;
    yPosition = y;
    ballColor = color(255, 255, 255);
    radius = 20;
  }
}

//add class definitions above this line

void setup() {
  size(400, 400);
}
```

The next step is to create the method `drawBall` which will draw the `Ball` object to the screen. This method does not have any parameters. Use the `ballColor` attribute with the `fill` command to color the ball. Then use the `circle` command to draw a circle. **Note** the `circle` method takes the x-position, y-position, and the diameter as arguments. Since the `Ball` class has the attribute `radius`, this attribute needs to be multiplied by 2.

```
  Ball(float x, float y) {
    xPosition = x;
    yPosition = y;
    ballColor = color(255, 255, 255);
    radius = 20;
  }

  void drawBall() {
    fill(ballColor);
    circle(xPosition, yPosition, radius * 2);
  }
```

Now that the `Ball` class has a method to draw a shape to the window, we are going to instantiate a `Ball` object. In Processing, global variables are declared before the `setup` method. Values are then given to these variables inside the `setup` method. Finally, call the `drawBall` method in `draw`. You should see a white circle appear in the window.

```
Ball ball;

void setup() {
  size(400, 400);
  ball = new Ball(50, 50);
}

void draw() {
  ball.drawBall();
}
```

Processing is drawing a black stroke around the circle. To remove this, use the command `noStroke();` in the `drawBall` method. Now you have a solid white circle (and the diameter is still 40 pixels); The `noStroke` command must come before `circle`. It does not matter if `noStroke` comes before or after `fill`.

```
  void drawBall() {
    noStroke();
    fill(ballColor);
    circle(xPosition, yPosition, radius * 2);
  }
```

▼ **Code**

```
//add class definitions below this line

class Ball {
  float xPosition;
  float yPosition;
  color ballColor;
  int radius;

  Ball(float x, float y) {
    xPosition = x;
    yPosition = y;
    ballColor = color(255, 255, 255);
    radius = 20;
  }
}

//add class definitions above this line

Ball ball;

void setup() {
  size(400, 400);
  ball = new Ball(50, 50);
}

void draw() {
  ball.drawBall();
}
```

Mutability Lab 3

Lab 3 - Animating the Ball

info

Animation File

For the next two pages, you are going to use the `animation.pde` file. In the Processing window, click `File` then `Open...`. On the left towards the bottom of the list, click `workspace`. Double click on `code`, double click on `mutability`, and double click on `animation`. Finally, open the `animation.pde` file.

Now that the ball appears on the screen, it is time to make the ball move. The movement should be two dimensional. That means `xPosition` and `yPosition` should both change over time. If the same change is applied to `xPosition` and `yPosition` equally, the ball will only move at a 45-degree angle. A more realistic animation will allow for a greater variation in movement. So two more instance attributes need to be added to the `Ball` class. One will control the velocity in the x-direction, and the other will control the velocity in the y-direction. Both of these attributes will be of type `float`.

```
//add class definitions below this line

class Ball {
  float xPosition;
  float yPosition;
  color ballColor;
  int radius;
  float xVelocity;
  float yVelocity;

  Ball(float x, float y) {
    xPosition = x;
    yPosition = y;
    ballColor = color(255, 255, 255);
    radius = 20;
    xVelocity = 1;
    yVelocity = 2;
  }
```

Next, there needs to be a new instance method that updates the position of the ball based on the newly created instance attributes. Create the method `updateBall`. It will add the x-velocity to the x-position, as well as add the y-velocity to the y-position.

```
  void drawBall() {
    noStroke();
    fill(ballColor);
    circle(xPosition, yPosition, radius * 2);
  }

  void updateBall() {
    xPosition += xVelocity;
    yPosition += yVelocity;
  }
```

Since the `draw` method is an infinite loop, `updateBall` should be called here. That way the ball's position is constantly being updated.

```
void draw() {
  ball.drawBall();
  ball.updateBall();
}
```

The ball moves, but it leaves a “trail” of where it has been. This happens because Processing is drawing each updated ball on the same background. Instead, we want to color the background (erase the trail), draw the ball, and then update its position. These three steps should happen each time the `draw` method runs. Add the `background` method **before** you draw the ball. Pass `background` an RGB value for the color of the window.

```
void draw() {
  background(55, 55, 55);
  ball.drawBall();
  ball.updateBall();
}
```

▼ **Why is the animation not always smooth?**
This has to do with how Processing works and how Codio was built. Processing uses something called X server to display graphical output. X server runs on a machine, not in a browser. Codio was designed to have a coding environment run in your browser. To get Processing output into your browser, X server is running on a server farm somewhere far away. Your Processing code gets sent to the server farm, where Processing output is generated, and then sent back to your browser. This means your program’s performance depends on network speeds. If your internet connection is not very good, or there is lots of network traffic, this will decrease the quality of your animation.

The animation should work, but the ball disappears off the screen. It is time to make the ball bounce. The general steps to getting ball to bounce are:

1. Use `xPosition` and `yPosition` and ask if the ball is at the edge of the window
2. If yes, then change direction of the velocity

Create the method `bounceBall` with two conditionals. The first checks if the ball is touching left or right sides of the screen, while the second conditional checks if the ball is touching the top or bottom. The edges of the screens are represented by a number between 0 and 400 (the width and height of the window). If the ball is touching the edge of the screen, its position is either less than 0 or greater than 400. When this happens, multiply the appropriate velocity attribute by -1. This will turn a positive number negative and a negative number positive.

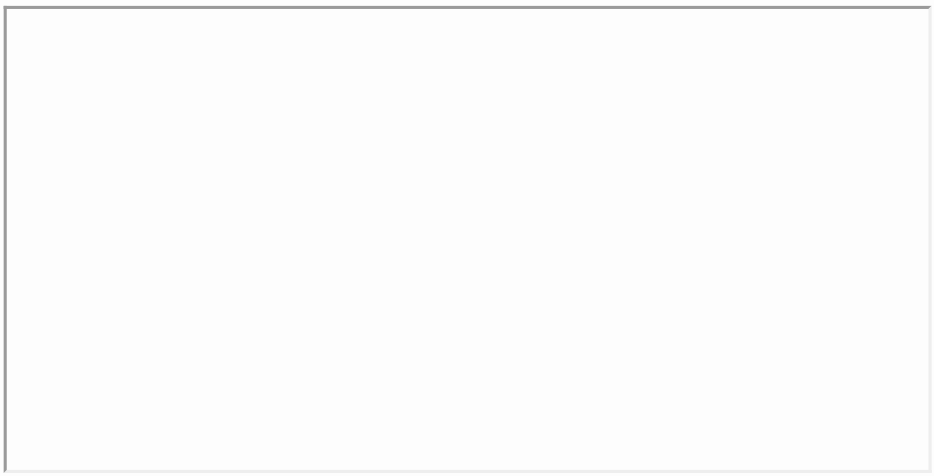
```
  void bounceBall() {
    if (xPosition < 0 || xPosition > 400) {
      xVelocity *= -1;
    }

    if (yPosition < 0 || yPosition > 400) {
      yVelocity *= -1;
    }
  }
```

Now call the `bounceBall` method in `draw`.

```
void draw() {
  background(55, 55, 55);
  ball.drawBall();
  ball.updateBall();
  ball.bounceBall();
}
```

The ball will bounce, but the animation will not look quite right. That is because the location of the circle is its center. So asking if `xPosition` is less than 0 (the left side of the window) means that this boolean expression will be true when half the circle is off of the window. A better way to build the animation is to ask if `xPosition` is less than `radius`. The animation on the left tests if the center of the ball is at the edge of the window. The animation on the right tests if the edge of the ball (the center plus or minus the radius) is touching the edge of the window.



▼ **Why is this animation much smoother than mine?**

The animation above is written in JavaScript. This language runs entirely in the browser. So internet connectivity or network traffic will not affect the animation once the JavaScript code has been downloaded to your

browser.

Update the `bounceBall` method so that it is asking if the center minus the radius is less than 0 or if the center plus the radius is greater than 400. The ball should now bounce when the edge of the ball touches the edge of the screen.

```
void bounceBall() {
  if (xPosition - radius < 0 || xPosition + radius > 400) {
    xVelocity *= -1;
  }

  if (yPosition - radius < 0 || yPosition + radius > 400) {
    yVelocity *= -1;
  }
}
```

You now have a complete bouncing ball animation.

▼ **Code**

```
//add class definitions below this line

class Ball {
  float xPosition;
  float yPosition;
  color ballColor;
  int radius;
  float xVelocity;
  float yVelocity;

  Ball(float x, float y) {
    xPosition = x;
    yPosition = y;
    ballColor = color(255, 255, 255);
    radius = 20;
    xVelocity = 1;
    yVelocity = 2;
  }

  void drawBall() {
    noStroke();
    fill(ballColor);
    circle(xPosition, yPosition, radius * 2);
  }
```

```
  void updateBall() {
    xPosition += xVelocity;
    yPosition += yVelocity;
  }

  void bounceBall() {
    if (xPosition - radius < 0 || xPosition + radius > 400) {
      xVelocity *= -1;
    }

    if (yPosition - radius < 0 || yPosition + radius > 400) {
      yVelocity *= -1;
    }
  }
}

//add class definitions above this line

Ball ball;

void setup() {
  size(400, 400);
  ball = new Ball(50, 50);
}

void draw() {
  background(55, 55, 55);
  ball.drawBall();
  ball.updateBall();
  ball.bounceBall();
}
```

Mutability Lab 4

Lab 4 - Tips and Tricks

info

Animation File

Finally, you are going to use the `animation.pde` file. In the Processing window, click `File` then `Open...`. On the left towards the bottom of the list, click `workspace`. Double click on `code`, double click on `mutability`, and double click on `animation`. Finally, open the `animation.pde` file.

Change Colors

One way to make the animation more interesting would be to have the ball change color each time it bounces. Since color is represented by three numbers, you could add to or subtract from these numbers. Doing so would only make minor changes to the color. Colors are represented by number between 0 and 255, so you have to think about what happens when the color values are smaller than 0 or greater than 255. A better way to implement this is to choose three random numbers each time the ball bounces.

Modify the `bounceBall` method so that the method `changeColor()` is called after the ball bounces. Be sure to add the method call to both conditionals. If not, the ball will only change color when it hits two of the four walls.

```
void bounceBall() {
  if (xPosition - radius < 0 || xPosition + radius > 400) {
    xVelocity *= -1;
    changeColor();
  }

  if (yPosition - radius < 0 || yPosition + radius > 400) {
    yVelocity *= -1;
    changeColor();
  }
}
```

Now, declare the `changeColor` method. To make the code easy to read, the variables `red`, `green`, and `blue` each get a random integer between 0 and 255. The `random` method returns a float between 0 and up to (but not including) the number in parentheses. `floor` is used to truncate a float into an int. These new values will be the new color. The ball should now change color every time it touches the edge of the window.

```
void changeColor() {
  int red = floor(random(256));
  int green = floor(random(256));
  int blue = floor(random(256));
  ballColor = color(red, green, blue);
}
```

Random Direction

The animation always starts in the same way. It would be more interesting if the ball moved in a randomly selected direction. Using the `random` method from above, create random values for `xVelocity` and `yVelocity` seems pretty easy; just use `random(-3, 3)`. However, there is a chance that a number close to 0 will be selected. That means the ball will move very slowly, and perhaps not move at all if both velocities are 0. What you really want to do is pick a random number between -3 and -1 or between 1 and 3. In the constructor, set the values of `xVelocity` and `yVelocity` to `randomVelocity`.

```
Ball(int x, int y) {
  xPosition = x;
  yPosition = y;
  radius = 20;
  ballColor = color(255, 255, 255);
  xVelocity = randomVelocity();
  yVelocity = randomVelocity();
}
```

There are two random decisions that need to be made. One, will the ball move in a positive or negative direction. Two, define that random number. Start with by asking if `random(1) < 0.5`. This means to take a random float from 0 up to but not including 1. If it is less then `0.5` the ball will travel in one direction; if it is greater it will travel in the other. If the conditional is true, choose a random number between 1 and 3. If false, choose a random number from -3 to -1. Every time you start the program, the ball should move in a randomly selected direction.

```
float randomVelocity() {
  if (random(1) < 0.5) {
    return random(1, 3);
  } else {
    return random(-3, -1);
  }
}
```

Avoiding Hard Coding

Assume we want to start the ball in the middle of the window when the animation starts. You could instantiate the `Ball` object like this.

```
ball = new Ball(200, 200);
```

The window is 400 by 400, so the middle would be position (200, 200). Change the window dimensions to 500 by 500, and the ball is no longer in the middle. That is because the starting position is hard coded into the program. That is, the starting position is a fixed number that is independent of the window dimensions. A better way to create the animation is to make the starting position dependent upon the window. Processing has two variables that you do no need to declare: `width` and `height`. These variables represent the width and height of the window. Divide them by 2 when instantiating the `Ball` object. The ball should always start in the middle of the window.

```
ball = new Ball(width / 2, height / 2);
```

challenge

Try these variations:

- If you change the window size, the ball no longer bounces when it touches the edge of the screen. Modify the `bounceBall` method so that it always bounces when it touches the edge of the screen.

▼ **Possible Solution**

The problem is that `400` is hard coded in this method. Replace these numbers with `width` (x-direction) and `height` (y-direction).

```
void bounceBall() {
  if (xPosition - radius < 0 || xPosition + radius >
      width) {
    xVelocity *= -1;
    changeColor();
  }

  if (yPosition - radius < 0 || yPosition + radius >
      height) {
    yVelocity *= -1;
    changeColor();
  }
}
```

- Add a method that increases the speed of the ball each time it bounces.

▼ **Possible Solution**

Note, the x-velocity increases when the ball hits the left or right sides of the window, and the y-velocity when the ball hits the top or bottom of the window. Call the `increaseVelocity` method after the ball bounces. Then create the `increaseVelocity` method and pass it `"x"` or `"y"` to increase the correct velocity. If the current velocity is negative, subtract 1. If the current velocity is positive, then add 1.

```
void bounceBall() {
  if (xPosition - radius < 0 || xPosition + radius >
      width) {
    xVelocity *= -1;
    changeColor();
    increaseVelocity("x");
  }

  if (yPosition - radius < 0 || yPosition + radius >
      height) {
    yVelocity *= -1;
    changeColor();
    increaseVelocity("y");
  }
}

void increaseVelocity(String direction) {
  if (direction.equals("x")) {
    if (xVelocity > 0) {
      xVelocity += 1;
    } else {
      xVelocity -= 1;
    }
  } else {
    if (yVelocity > 0) {
      yVelocity += 1;
    } else {
      yVelocity -= 1;
    }
  }
}
```

- Add a method that makes the ball grow each time it bounces.

▼ **Possible Solution**

Call the `growBall` method after the ball bounces. Then define this method to increase the `radius` attribute by 1.

```
  void bounceBall() {
    if (xPosition - radius < 0 || xPosition + radius >
        width) {
      xVelocity *= -1;
      changeColor();
      increaseVelocity("x");
      growBall();
    }

    if (yPosition - radius < 0 || yPosition + radius >
        height) {
      yVelocity *= -1;
      changeColor();
      increaseVelocity("y");
      growBall();
    }
  }

  void growBall() {
    radius += 1;
  }
```

▼ **Code**

```
  //add class definitions below this line

  class Ball {
    float xPosition;
    float yPosition;
    color ballColor;
    int radius;
    float xVelocity;
    float yVelocity;

    Ball(float x, float y) {
      xPosition = x;
      yPosition = y;
      ballColor = color(255, 255, 255);
      radius = 20;
      xVelocity = randomVelocity();
      yVelocity = randomVelocity();
    }

    void drawBall() {
      noStroke();
      fill(ballColor);
      circle(xPosition, yPosition, radius * 2);
```

```
}

void updateBall() {
  xPosition += xVelocity;
  yPosition += yVelocity;
}

void bounceBall() {
  if (xPosition - radius < 0 || xPosition + radius > width)
    {
    xVelocity *= -1;
    changeColor();
    increaseVelocity("x");
    growBall();
  }

  if (yPosition - radius < 0 || yPosition + radius > height)
    {
    yVelocity *= -1;
    changeColor();
    increaseVelocity("y");
    growBall();
  }
}

void growBall() {
  radius += 1;
}

void increaseVelocity(String direction) {
  if (direction.equals("x")) {
    if (xVelocity > 0) {
      xVelocity += 1;
    } else {
      xVelocity -= 1;
    }
  } else {
    if (yVelocity > 0) {
      yVelocity += 1;
    } else {
      yVelocity -= 1;
    }
  }
}

  void changeColor() {
    int red = floor(random(256));
    int green = floor(random(256));
    int blue = floor(random(256));
```

```
    ballColor = color(red, green, blue);
  }

  float randomVelocity() {
    if (random(1) < 0.5) {
      return random(1, 3);
    } else {
      return random(-3, -1);
    }
  }
}

//add class definitions above this line

Ball ball;

void setup() {
  size(500, 500);
  ball = new Ball(width / 2, height / 2);
}

void draw() {
  background(55, 55, 55);
  ball.drawBall();
  ball.updateBall();
  ball.bounceBall();
}
```

Mutability Lab Challenge

Copy and paste the `Zoo` class below into the code editor.

```
//add class definitions below this line

class Zoo {
  int bigCats;
  int primates;
  int reptiles;
  int birds;

  Zoo(int bc, int p, int r, int b) {
    bigCats = bc;
    primates = p;
    reptiles = r;
    birds = b;
  }
}
//add class definitions above this line
```

Add the following methods to the class:
* `totalAnimals` - returns the total number of animals
* `totalMammals` - returns the number of mammals
* `mostAnimals` - returns the name with the most animals in the zoo

Expected Output

If the following code is added to your program:

```
    //add code below this line

    Zoo myZoo = new Zoo(10, 30, 90, 120);
    System.out.println(myZoo.totalAnimals());
    System.out.println(myZoo.totalMammals());
    System.out.println(myZoo.mostAnimals());

    //add code above this line
```

Then the output would be:

```
250
40
birds
```