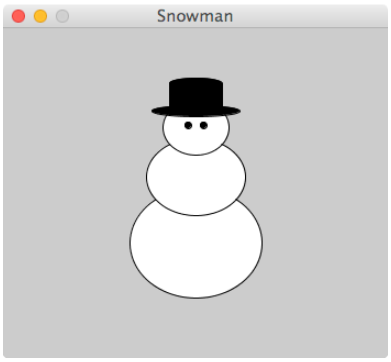

CIS 287 — First Individual Programming Assignment

Part 1

Go to https://processing.org and download the latest version (2.2.1 as of January 2015) of Processing appropriate for your operating system. Unzip it, navigate to the executable (the file with the Processing icon), and enter the program below.

```
ellipse(25, 75, 15, 25);
```

Run the program by pushing the "play button" at the top of the window. Assuming it works, can you figure out the significance of the four numbers? (Make changes and see what happens.) How wide is the window? How tall? Where is (0, 0)?

Now try this:

```
size(320, 240);
fill(0);
rectMode(CENTER);
rect(width / 2, height / 2, 40, 50);
fill(255);
ellipseMode(RADIUS);
ellipse(width / 2, height / 2, 20, 25);
```

Does what you see make sense to you? You can look up Processing's built-in functions, global variables (e.g., `width`), and constants here: https://processing.org/reference/.

What's the difference between `fill(0);` and `fill(255);`? What happens if you comment out (with `//`) the line `rectMode(CENTER);`? What happens if you comment out `ellipseMode(RADIUS);`?

Try to make a snowman that looks like this:

If you want to add arms and a mouth, look up `line` and `arc` in the Processing documentation. Save your program as "Snowman" (Snowman.pde). You'll submit this as the first part of the assignment.

Part 2

Open a new window; enter and run this program:

```
void setup() {
  size(320, 240);
  background(255);
}
```

```
void draw() {
  fill(mouseY, 20, 20);
  ellipse(mouseX, height / 2, 100, 100);
}
```

`setup` and `draw` are special functions, called automatically by the Processing interpreter (assuming you've defined them). When are they called? (Hint: Try moving `background(255);` from `setup` to `draw` and see what happens.) Also notice that `fill` is called with three arguments—the red, green and blue components of a color. What is the minimum value that works? The maximum?

Modify your program to make it a liitle more interesting:

```
int oldX, oldY;

void setup() {
  size(320, 240);
  background(255);
  oldX = mouseX;
  oldY = mouseY;
}

void draw() {
  fill(mouseY, min(mouseX, 255), -mouseY);
  ellipse(oldX, oldY, 40, 40);
  oldX = (oldX * 9 + mouseX) / 10;
  oldY = (oldY * 9 + mouseY) / 10;
}
```

Run the program. Do you understand how it works? Experiment with the code, making changes to lines you aren't sure about, and see what happens.

Hint: The modified version of `draw` below shows more clearly that `oldX`'s new value is based on a weighted average of its old value and the value of `mouseX`. (Same idea with `oldY`.)

```
void draw() {
  fill(mouseY, min(mouseX, 255), -mouseY);
  ellipse(oldX, oldY, 40, 40);

  int oldWeight = 9;
  int newWeight = 1;

  oldX = (oldX * oldWeight + mouseX * newWeight) /
      (oldWeight + newWeight);
  oldY = (oldY * oldWeight + mouseY * newWeight) /
      (oldWeight + newWeight);
}
```

When you feel confident that you understand this program as it is, modify it to make your own interesting interactive design. Be creative! Save your program as "Circles" (hopefully that name will still make sense); you'll submit it for part 2 of this assignment.

Part 3

The next program introduces `translate` and `scale`, tools for changing the coordinate system to make it suit the program you're working on.

```
void setup() {
  size(512, 512);
  strokeWeight(0.5);
}

void draw() {
  background(0);

  translate(mouseX, mouseY);
  scale(width / 10, -width / 10);

  stroke(240, 40, 40);
  line(0, 0, 1, 1);

  stroke(240, 160, 40);
  line(0, 0, 1, -1);

  stroke(240, 240, 40);
  line(0, 0, -1, -1);

  stroke(80, 220, 40);
  line(0, 0, -1, 1);
}
```

Notice that all four lines begin at (0, 0), but the location of (0, 0), changes as the mouse moves. Which line changes it? Also notice that the negative y direction in this program is down, not up (as it was in the last two programs). Can you figure out how to change it back, so that the negative y direction is up? Experiment with the calls to `translate` and `scale`, changing the values, predicting the effect of the change, and then running the program to see if your prediction was correct.

Once you understand how `translate` and `scale` work, use them to make a new version of your snowman program, in which the snowman follows the mouse around the window. Save your program as "MovingSnowman" and submit it for part 3 of this assignment.

Part 4

Next, try this program. You don't need to retype all of the comments I've used to explain the details, but you should type the Processing statements (rather than copy / pasting) and try to understand what each one does, even if you aren't sure yet how that statement fits into the overall program.

```
float angle = 0;  // angle will be a floating-point (not an
                  // integer) variable.
void setup() {
  size(640, 480);
  strokeWeight(0.01);
  noLoop();  // draw will be called once, rather than
             // being called again and again automatically.
             // If the scene should be redrawn, the redraw
             // function should be called.
}

void draw() {
  background(0);
  int r = int(random(0, 256));  // random will return a float
                                // value between 0 and 256
                                // (including 0 through 255 as
                                // possibilities).  stroke needs
```

```
                                // an int value; the int function
                                // is used to convert it from
                                // float to int.
  stroke(r);

  translate(width / 2, height / 2);
  scale(width / 4, -width / 4);

  rotate(angle);  // Like translate and scale, rotate can be
                  // used to change the coordinate system. A
                  // positive angle rotates the x and y axes
                  // counterclockwise.

  line(-1, -1, 1, 1);
  rotate(angle * 2);
  line(-1.0 / 2, -1.0 / 2, 1.0 / 2, 1.0 / 2);  // 1.0, not 1, used
                                                // to get floating-
                                                // point, not integer
                                                // division.
  rotate(angle * 3);
  line(-1.0 / 3, -1.0 / 3, 1.0 / 3, 1.0 / 3);
}

void keyPressed() {

  if (keyCode == UP) {
    angle += 0.1;  // Equivalent to:  angle = angle + 0.1;

    redraw();  // Proper way to get the code in draw to run
               // again when noLoop has been used to turn off
               // repeated calls to draw. (draw shouldn't be
               // called directly.)

  } else if (keyCode == DOWN) {
    angle -= 0.1;  // Equivalent to:  angle = angle - 0.1;
    redraw();
  }
}
```

Run the program to see what it does. (Hold down the up arrow key for a while, then hold down the down arrow key for a while...) Now reread the program, thinking about how each statement fits into the overall program. Is it clear to you when `setup`, `draw` and `keyPressed` are being called? Experiment with different colors, line widths, and angle changes. Each time you make a change, predict what the result of the change will be. Then run the program to see if your prediction was correct.

Wouldn't it be cool if there were an easy way to make more than three lines? There is! You can use a loop:

```
for (int i = 1; i <= 10; i++) {
  rotate(angle * i);
  line(-1.0 / i, -1.0 / i, 1.0 / i, 1.0 / i);
}
```

The first time through the loop, `i` is 1; after each iteration the value of `i` is increased by 1. (That's what `i++` means—increase the value of `i` by 1.) The loop continues until the condition `i <= 10` becomes false. So the code inside the loop runs 10 times, with the values 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 assigned to `i`.

Try adding a function called `drawStar` with the loop...

```
void drawStar() {

  for (int i = 1; i <= 10; i++) {
    rotate(angle * i);
    line(-1.0 / i, -1.0 / i, 1.0 / i, 1.0 / i);
  }
}
```

...now add statements to call `drawStar` from `draw`, use `translate` and `scale` to change the coordinate system, and then call `drawStar` again (to get a second star in a different place):

```
void draw() {
  background(0);
  int r = int(random(0, 256));
  stroke(r);

  translate(width / 2, height / 2);
  scale(width / 4, -width / 4);

  drawStar();  // Original star in the center.

  translate(-1, 0);  // Move origin one unit left.

  scale(0.5, 0.5);  // Make everything 1/2 the size.

  drawStar();  // Draw a 1/2 size star one unit left
               // of the original.
}
```

At first it probably looks the way you expect it to, but try hitting the up or down arrow keys. It's cool! But it's not what you'd expect. Why is the second star orbiting around the first? It's because the rotations in the first call to `drawStar` change the effect of `translate(-1, 0);`—the coordinate system is getting rotated, so the negative x direction is changing before `translate` is called. (Why wasn't this a problem until now? Because the coordinate system is reset automatically before each time `draw` runs.)

Wouldn't it be cool if there were an easy way to reset the coordinate system to what it was before all those rotations? There is! Use the `pushMatrix` function to save a copy of the current coordinate system. ("Push" because you are putting the copy on the top of a "stack" data structure; "matrix" because `translate`, `scale` and `rotate` are implemented with matrix multiplication—but you don't really need to understand how it all works to use them.) ...And use `popMatrix` to restore the coordinate system to whatever it was the last time you called `pushMatrix`:

```
// STOP!  Did you get lost in the previous two paragraphs and
// skim down to this code snippet?  Go back and read...make
// sure you understand what's going on here!  :)

void drawStar() {
  pushMatrix();  // Save a copy of the coordinate system,
                 // whatever it was right when the function was
                 // called.

  for (int i = 1; i <= 10; i++) {
    rotate(angle * i);
    line(-1.0 / i, -1.0 / i, 1.0 / i, 1.0 / i);
  }

  popMatrix();  // Restore the coordinate system to what it was
```

```
                // when pushMatrix was called at the beginning
                // of this function.
}
```

Based on what you've read, and what you see in these examples, modify your program so that it puts two spinning stars side-by-side in the window. Make them spin in opposite directions. (Can you figure out a way to do it without changing the `drawStar` function?) Call your program "SpinningStars" and submit it for part 4 of this assignment.

Part 5

Another program for you to enter and run...this one introduces *arrays*—special variables that provide a way to access multiple related values of the same type (via just one variable).

```
int[] intArray;  // Declare the array variable.

void setup() {
  intArray = new int[5];  // Create an array and assign it to
                          // the array variable.

  for (int i = 0; i < 5; i++) {  // Assign values to the elements
    intArray[i] = 0;             // of the array.
  }

  size(700, 300);
  strokeWeight(0.1);
  ellipseMode(RADIUS);
  noLoop();
}

void draw() {
  background(255);
  translate(width / 2, height / 2);
  scale(width / 7, -width / 7);
  translate(-2, 0);

  for (int i = 0; i < 5; i++) {
    fill(intArray[i]);
    ellipse(i, 0, 0.4, 0.4);
  }
}

void keyPressed() {

  if (key == '1') {

    if (intArray[0] == 0) {  // Access a value in the array.

      intArray[0] = 255;     // Change a value in the array.

    } else if (intArray[0] == 255) {
      intArray[0] = 0;
    }

    redraw();
```

```
  } else if (key == '2') {

    if (intArray[1] == 0) {
      intArray[1] = 255;
    } else if (intArray[1] == 255) {
      intArray[1] = 0;
    }

    redraw();

  } else if (key == '3') {

    if (intArray[2] == 0) {
      intArray[2] = 255;
    } else if (intArray[2] == 255) {
      intArray[2] = 0;
    }

    redraw();

  } else if (key == '4') {

    if (intArray[3] == 0) {
      intArray[3] = 255;
    } else if (intArray[3] == 255) {
      intArray[3] = 0;
    }

    redraw();

  } else if (key == '5') {

    if (intArray[4] == 0) {
      intArray[4] = 255;
    } else if (intArray[4] == 255) {
      intArray[4] = 0;
    }

    redraw();
  }
}
```

This program uses an array variable to hold 5 integers. (I called the variable `intArray`, but you can call it whatever you want, just like a normal variable.) There are four steps to preparing and using an array that are illustrated by this program:

1. Declare the array variable:

   ```
   int[] intArray;
   ```

2. Create an array and assign it to the variable:

   ```
   intArray = new int[5];
   ```

3. Assign values to the elements of the array:

   ```
   for (int i = 0; i < 5; i++) {
     intArray[i] = 0;
   }
   ```

4. Access values previously assigned to elements of the array (and change them):

```
if (intArray[0] == 0) {  // Access a value.

  intArray[0] == 255;  // Change a value.
}
```

Notice that the first element of the array has index 0 (and the last has an index 1 less than the number of elements in the array).

There are several ways this program might be improved. First, since there are only two possible values for each element of the array, it would make sense to use an array of `boolean` (`true` or `false`) values rather than an array of integers. Also, it's possible to declare the array variable, create a new array, and assign values to each element...all in a single statement:

```
boolean[] boolArray = { false, false, false, false, false, false };
```

Wait a minute...if you count the number of elements (each `false` in the statement above) there's 6, not 5. So if you wanted to change the program to use `boolArray` instead of `intArray`, you'd have to make it use 6 wherever 5 was used in the original version. There's a better way to deal with that problem, however. Use the `length` field created automatically for array variables:

```
translate(width / 2, height / 2);
float scaleFactor = width / (boolArray.length + 2);
scale(scaleFactor, -scaleFactor);
translate(-(boolArray.length - 1) / 2.0, 0);
```

These lines 1) put the origin at the center of the window, 2) scale the window so that, if there are n elements in the array, the window is $n+2$ units wide, and 3) move the origin to the proper position for the center of the leftmost circle. Most importantly, they will work no matter how many elements are in the array. The array's `length` field can also be used to make a loop work for any size array:

```
for (int i = 0; i < boolArray.length; i++) {

  if (boolArray[i]) {  // Same as:  if (boolArray[i] == true) {
    fill(255);
  } else {
    fill(0);
  }
}
```

Notice how the code inside the loop changes when the array holds `boolean` rather than `int` values. Since a `boolean` value is either `true` or `false`, it can be used as a `boolean` expression in an `if` statement.

Finally, `keyTyped` can be improved a lot—it's much longer in the code above than it needs to be. Consider the following statement:

```
int index = key - '1';
```

`key` is a `char` (for "character") variable. A `char` variable is actually a special kind of integer, whose value can be interpreted as a number or as a character (the character assigned to that number in the ASCII standard). `'1'` is a `char` too: it can be interpreted as the character "1" or the number 49; `'2'` is a `char` that can be interpreted as "2" or 50; `'3'` can be interpreted as "3" or 51, etc. The expression `key - '1'`, then, will have the value 0 if `key` is `'1'`, 1 if `key` is `'2'`, 2 if `key` is `'3'`, and so on. This value is the exactly what we need! We can use it to access the array element that goes with each number on the keyboard.

Once we have this index value, we can check if it's a valid index for our array. If it's not valid, the user hit some other key, which should be ignored. If it is valid, we can toggle the value at that index (make it `true` if it was `false`, or `false` if it was `true`), and then update the window:

```
if (index >= 0 && index < boolArray.length) {
  boolArray[index] = !boolArray[index];
  redraw();
}
```

(Can you figure out what && and ! mean? Check the Processing documentation online if you aren't sure.)

Here's the full program with all of these changes.

```
boolean[] boolArray = { false, false, false, false, false, false };

void setup() {
  size(700, 300);
  strokeWeight(0.1);
  ellipseMode(RADIUS);
  noLoop();
}

void draw() {
  background(255);
  translate(width / 2, height / 2);
  float scaleFactor = width / (boolArray.length + 2);
  scale(scaleFactor, -scaleFactor);
  translate(-(boolArray.length - 1) / 2.0, 0);

  for (int i = 0; i < boolArray.length; i++) {

    if (boolArray[i]) {
      fill(255);
    } else {
      fill(0);
    }

    ellipse(i, 0, 0.4, 0.4);
  }
}

void keyPressed() {
  int index = key - '1';

  if (index >= 0 && index < boolArray.length) {
    boolArray[index] = !boolArray[index];
    redraw();
  }
}
```

For more practice with arrays, modify this program so that, instead of hitting a number key to lighten or darken a circle, the arrow keys are used to select a circle and the space bar lightens or darkens it. To mark a circle selected, call `stroke` with values for a lighter color; leave the outline black for non-selected circles. Also, you can use `key == ' '` to check whether the user hit the space bar.

Save your modified program as "PickACircle" and submit it for part 5 of this assignment.

Part 6

For part 6 you'll create an interactive Tic Tac Toe program—your first game! But you need to know about "two-dimensional" arrays before you start...

In Processing you can make an array to hold any kind of values, which means you can make an array of arrays. This is a common thing to do, so there's a special syntax for it:

1. To declare a two-dimensional array variable (to hold `char` values, in this case):

```
char[][] cells;
```

2. To create a two-dimensional array and assign it to `cells`:

```
cells = new char[N][N];  // N would be 3 for a normal
                         // Tic Tac Toe board.
```

3. To assign values to each element of the array, you can use a nested loop:

```
for (int x = 0; x < cells.length; x++) {
  for (int y = 0; y < cells[x].length; y++) {
    cells[x][y] = ' ';
  }
}
```

4. To access values previously assigned to elements of the array (and change them):

```
if (cells[0][0] == ' ') {
  cells[0][0] = 'X';
}
```

Notice how `cells.length` and `cells[x].length` are used to make the nested loop work for any size two-dimensional array. `cells.length` is the number of sub-arrays; the sub-array at index `x` is `cells[x]`, and so `cells[x].length` is the length of the sub-array at index `x`.

Start your Tic Tac Toe program with the following code, which draws a 3 × 3 board:

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

void draw() {
  translate(0, height);          // Move origin to lower left.
  scale(width / 5, -width / 5);  // Make window 5x5, flip y axis.
  translate(1, 1);               // Move origin up 1 and over 1
                                 // (where we'll put the lower
                                 // left corner of the board).
  strokeWeight(0.05);
  background(255);
  drawBoard();
}

void drawBoard() {
  stroke(100);

  for (int i = 1; i < 3; i++) {
    line(i, 0, i, 3);  // vertical
    line(0, i, 3, i);  // horizontal
  }
}
```

Modify the program so that it works for any size (square, at least 2×2) board. Then add code to draw a light-colored square initially filling the lower left space on the board; make it so that, if the user hits an arrow key, the lighter square moves to an adjacent space, as long as this won't move it off the board.

If you aren't sure what to do, here's (roughly) how my program works:

1. At the beginning of the program I declare two variables, `xMarker` and `yMarker`, to keep track of where the lighter square should be. Both `xMarker` and `yMarker` are initialized to zero.
2. I have a `drawMarker` function that draws the ligher square, a 1×1 rectangle whose lower left corner is at (`xMarker`, `yMarker`). I call this function in `draw`, before calling `drawBoard`, so that the grid lines show over the marker square.
3. I have a `keyPressed` function that changes the value of `xMarker` or `yMarker`, if the user hits one of the arrow keys, and then calls `redraw` to update the window. It's in this function that I check to see if changing the value of `xMarker` or `yMarker` would move the marker square off the board. If it would, that key press is ignored.

Once you have this working, add code so that, if the user hits the space bar when the marker square is over an empty space on the board, either an "X" or "O" is drawn in the space. To do this, you'll need 1) a variable to keep track of whose turn it is (alternating between "X" and "O"), and 2) a two-dimensional array to keep track of whether each board space is blank, contains "X", or contains "O."

Here's how my program does all this:

1. At the beginning of the program I declare a global `char` variable, called `turn`, and set it to `'X'`. I also declare a (global) two-dimensional array of `char` values, called `cells`, and assign a new $N \times N$ `char` array to it. (Refer to the beginning of this section, part 6, if you aren't sure how to do this.)
2. In `setup`, I use a nested loop to fill the two-dimensional array `cells` with `' '` (the space character). (Refer to the beginning of this section...)
3. In `keyPressed`, I added a case to the `if` statement to respond when the user hits the space bar. If they hit the space bar, I check the two-dimensional array `cells` at indices `xMarker` and `yMarker` to see if that space is blank (i.e., it's `' '` rather than `'X'` or `'O'`). If it is, I call the `updateCells` function (described next) and then call `redraw` to update the window.
4. In the `updateCells` function, I set the cell at indices `xMarker` and `yMarker` (in the two-dimensional array `cells`) equal to the current value of `turn`. Then I change the value of `turn`—to `'O'` if it was `'X'`; to `'X'` if it was `'O'`.
5. Finally, I have a `drawCells` function that loops through the two-dimensional array `cells`, drawing an "X" or an "O" (or leaving the space blank) on the board, depending on what it finds in the array. This function is called from `draw`, after `drawMarker`, so that the "X" or "O" for a filled square appears in front of the marker square. Here's pseudocode for `drawCells`:

```
for x in 0...cells.length
  for y in 0...cells[x].length
    save coordinate system
    move origin to (x, y)

    if cells[x][y] is 'X'
      draw an X (two lines)
    else if cells[x][y] is 'O'
      draw an O (a circle)

    restore saved coordinate system
```

Once you have all of this working, tweak it to make it look nice (make it colorful, etc.). Also test it to make sure it works for any size (square, at least 2×2) board. Save your program as "TicTacToe" and submit it for part 6.

Part 7 (Optional Extra Challenge)

For an optional extra challenge, create a computer opponent for the user to play against. Start by designing a strategy; then break up the implementation of the strategy into a series of development steps; then implement and test them one at a time. For example, here's what I tried:

1. Strategy:

   - If you can win in one move, win.
   - Otherwise, if the other player can win in one move, block.
   - Otherwise, pick a random space and play in it.

   (This turns out to be a decent strategy for preventing the other player from winning, but not a great strategy for the computer to actually win games.)

2. First development step:

   - Make the computer play in a random (available) square.

3. Next step:

   - Make the computer play in the first available square that completes a row or column. (This could be a win or a block.)
   - If no such square is available, fall back to the previous step—play in a random square.

4. Next step:

   - Distinguish between wins and blocks; make the computer try to win first and only block if it can't win.
   - If it can't win or block, it should play in a random square.

5. Next step:

   - Add code to handle diagonal wins and blocks...

6. Final step:

   - Detect wins.
   - Don't allow either player to play after one of them has won.

If you complete this version of the program, save it as "TicTacTwo" (isn't that clever!) and submit it for part 7.

Grading

Your grade will be based on the following rubric:

/ 10	Snowman.pde submitted, shows reasonable effort, runs without errors to produce nice-looking output.
/ 15	Circles.pde submitted, shows reasonable effort, runs without errors, includes creative modifications.
/ 10	MovingSnowman.pde submitted, shows reasonable effort, runs without errors, works correctly.
/ 10	SpinningStars.pde submitted, shows reasonable effort, runs without errors, works correctly.
/ 20	PickACircle.pde submitted, shows reasonable effort, runs without errors, works correctly.
/ 25	TicTacToe.pde submitted, shows reasonable effort, runs without errors, works correctly, produces nice-looking output.
/ 10	Randomly selected source code file has clear and consistent formatting, appropriate comments, etc.
/ 10	Up to 10 points *extra credit* for TicTacTwo.pde, although overall grade for the assignment may not exceed 100.