UNIT 13. ADVANCED LOOPS

Summary
You will see in this unit how loops scale up to solve more complex problems. In this section, you will…

- Work through a range of problems that require loops
- See how loops can nest inside other loops

Learning Objectives
After finishing this unit, you will be able to …

- Use loops to solve a wide range of problems
- Create a nested loop to iterate two variables over a range of numbers

How to Proceed
- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

 © James Young, 2016

13.1 **Introduction**

Loops seem to be one of the primary challenges faced by new programmers. The previous chapter introduced you to basic loops and, for the most part, you have seen all the syntax and mechanics that you need. However, where students often struggle is in the application of loops. Even though you may understand the mechanics, it is not obvious to see how to use loops to solve problems, or how loops scale up. For example, what happens if you place a loop inside a loop?

This unit is primarily practice and expansion of the previous unit. However, this doesn't mean that you can gloss over it. Working through for loop examples is a great way to bring all of what you learned together, and to practice your skills on more advanced problems. This is why these two units are so long, because it is very important.

First, let's start with a larger example, and then get into new techniques.

13.2 **Example: Clickable Calendar**

We have now learned enough tools to make a simple clickable calendar. As always, a great place to start is by defining a bunch of constants that help define your calendar, at the top of your program. In this case, we need to determine where the calendar is located on the screen, how many days are in the calendar, and the spacing between the days (the grid). These are the numbers that I used:

```
final int CAL_TOP = 100;
final int CAL_LEFT = 100;
final int CAL_DAYS = 31;
final int CAL_SPACE = 30;
```

Next, let's draw the header row – the titles for the days of the week. You can make this look nicer by putting a background rectangle or changing color, but for simplicity we don't do that here. As always again, helper variables make our thinking a lot easier. Let's set a variable for the left and the bottom of the row – remember that graphical text wants the `x,y` for the BOTTOM LEFT of the text. So, we add one spacing to the top of the calendar to get the bottom of the first row.

```
int bottom = CAL_TOP+CAL_SPACE;
int left = CAL_LEFT
text("S", left, bottom);
```

 © James Young, 2016

To calculate the position for the next letter, M, we just add the space to the left variable and draw again:

```
left += CAL_SPACE;
text("M", left, bottom);
```

and so on. Draw all the letters this way. Unfortunately this is a little tedious. The good news is that we'll learn techniques later to simplify this kind of operation.

Next, let's draw the calendar numbers using a `for` loop (why not a `while` loop?). Use a single for loop through all the days of the month. Then, for each day, we can calculate which row and column it is on.

```
for (int day = 1; day <= CAL_DAYS; day++)
{
  // calculate row, column
}
```

To calculate the row and column we can just use integer math. This is a little tricky, but try to work through it. If you divide the day number by 7, it tells you which week you are on (with 0 as the first week). If you take the remainder, it tells you which day of the week (column, with 0 as the first one). This is confusing, so look at this table:

day	d/7	d%7		d/7	d%7		d/7	d%7
1	0	1	**11**	1	4	**21**	3	0
2	0	2	**12**	1	5	**22**	3	1
3	0	3	**13**	1	6	**23**	3	2
4	0	4	**14**	2	0	**24**	3	3
5	0	5	**15**	2	1	**25**	3	4
6	0	6	**16**	2	2	**26**	3	5
7	1	0	**17**	2	3	**27**	3	6
8	1	1	**18**	2	4	**28**	4	0
9	1	2	**19**	2	5	**29**	4	1
10	1	3	**20**	2	6	**30**	4	2

It just works out so nicely! So the row is the day / 7, and the column is the remainder. Actually, we need to add 1 to our row since we already have the header on row 0.

```
int col = i%7;
int row = i/7+1;
```

Now we use that information to calculate the left and bottom location to draw our text. I will add a helper variable in here called `top` since it will help out later.

```
left = CAL_LEFT + col*CAL_SPACE;
int top = CAL_TOP + row*CAL_SPACE;
```

 © James Young, 2016

```
bottom = top + CAL_SPACE;
text(str(day), left, bottom);
```

Now we have a nice calendar! The next step is to let the user select a day and to highlight the day that is selected. We can highlight it just by drawing a square as in our calendar image.

To do this, we need a global state variable that remembers or knows which square to highlight. For testing, let's just highlight day 5 by default:

```
int selected = 5; // day currently selected
```

Then, in our draw loop, we need to check which day we are drawing. If we are drawing the selected day, then we put a highlighted square around it. In my case, I set the colors, too.

```
if (day == selected)
{
  rect(left,right,CAL_WIDTH,CAL_WIDTH);
}
```

This only draws a square behind the selected day. Be careful of draw order! Now, how do we change the selected day based on where the user clicks? How can we detect if the mouse is clicked on a particular calendar square? Well, we already know the left, bottom, and top of each calendar square. If we calculate the right side, then we have all the information we need to do hit testing.

Further, instead of doing the match to calculate where the mouse is, we can do it inside the `for` loop on each day. On each day, let's test if the mouse hit it, and if so, update our selected day.

```
int right = left+CAL_SPACE;
if (mousePressed &&
  mouseX >= left && mouseX < right &&
  mouseY >= top && mouseY < bottom)
{
   selected = day;
}
```

Now as the user clicks on a day, it gets selected!

There are two more things we should do. First, the calendar should default to not having any day selected. The way we do this is to set our selected initially to an impossible day, so that our `if` statement comparing to the day never gets true. 0 is

 © James Young, 2016

an impossible day, so change the global definition to

```
int selected = 0; // impossible day, nothing selected
```

The final thing to do is to enable the person to *unselect* the day. How about, if the person clicks but they do not click on the calendar, then we unselect the day? How can we do this?

This is a little tricky. What we can do is this. Before going through drawing the calendar and checking every single day if it was clicked, we create a boolean that tells us if a day has been hit or not. Let's default it to false, as no day was hit.

```
boolean hit = false;
```

Then, as we go through the calendar, if we ever get a hit on a day, we set that boolean to true. Finally, after we check all the days, if we got NO hit, but the mouse is pressed, then we unselect the day. That's it! We now have a fully interactive calendar. A lot of new things popped up in this example, so be sure to study it. Here is my final code:

```
final int CAL_TOP = 100;
final int CAL_LEFT = 100;
final int CAL_DAYS = 31;
final int CAL_SPACE = 30;
int selected = 0;
void setup()
{
  size(500, 500);
}

void draw()
{
  background(0);
  fill(255);
  // draw title bar
  int bottom = CAL_TOP+CAL_SPACE;
  int left = CAL_LEFT;
  text("S", left, bottom);
  left += CAL_SPACE;
  text("M", left, bottom);
  left += CAL_SPACE;
  text("T", left, bottom);
```

 © James Young, 2016

```
left += CAL_SPACE;
text("W", left, bottom);
left += CAL_SPACE;
text("R", left, bottom);
left += CAL_SPACE;
text("F", left, bottom);
left += CAL_SPACE;
text("S", left, bottom);
left += CAL_SPACE;

boolean hit = false;
for (int day = 1; day<= CAL_DAYS; day++)
{
  int col = day%7;
  int row = day/7+1;

  left = CAL_LEFT+CAL_SPACE*col;
  int right = left+CAL_SPACE;
  int top = CAL_TOP+CAL_SPACE*row;
  bottom = top+CAL_SPACE;

  if (mousePressed &&
    mouseX >= left && mouseX < right &&
    mouseY >= top && mouseY < bottom)
  {
    selected = day;
    hit = true;
  }

  if (selected == day)
  {
    rect(left, top, CAL_SPACE, CAL_SPACE);
    fill(0);
  } else
  {
    fill(255);
  }
  text(str(day), left, bottom);
}
if (mousePressed && !hit)
```

 © James Young, 2016

```
  {
    selected = 0;
  }
}
```

To expand this example, try setting which day of the week the calendar starts on, or fixing the highlighting to be better centered. Additionally, how would you highlight weekend days?

13.3 **Nested Loops**

Loops can be tricky. As I said before, an important element of really getting loops is to memorize the mechanics – what happens at what stage. This is particularly important as we move into nested loops: you can put a loop inside another loop.

In a previous example, we drew a grid of lines using a `for` loop – we could iterate across the gaps, and draw one line at each gap. For a 10x10 grid, this took 22 lines. What if we wanted to draw a grid of small dots? Let's say we want a grid of 10 by 10 dots, which requires 100 dots. How would you go about doing it?

You could do it with one for loop, if you iterate `i` from 1...100, and then determine the grid coordinates depending on the `i`. This is similar to the calendar example above. Perhaps a more straightforward method is to use a nested for loop.

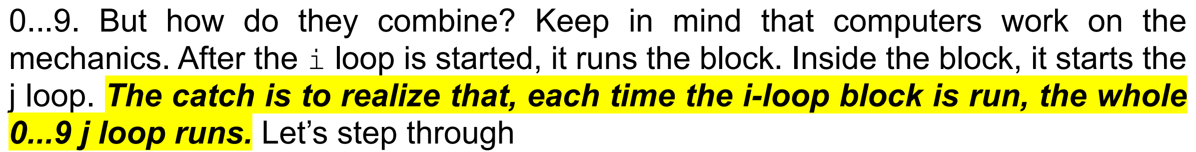
Note: since a code block acts like any other code, you can put a loop or if statement inside of any loop or if statement. Nested just means one thing inside another.

What happens if we put one for loop inside another? There is nothing special here, all the same rules of programming apply. However, it becomes a little tricky to keep in your head. Look at the following code:

```
int count = 0;
for (int i = 0; i < 10; i++)
{
      for (int j = 0; j < 10; j++)
      {
            count++;
      }
}
println(count);
```

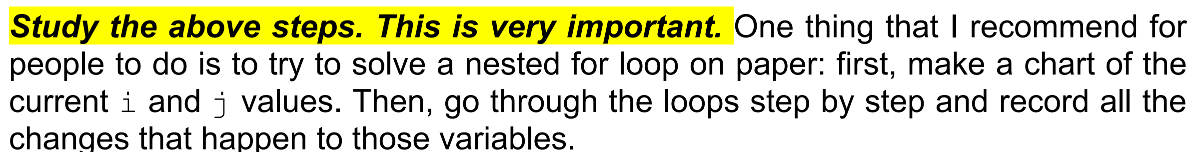
Both of these for loops should make sense to you. The first one iterates `i` from 0...9 (remember – the loop quits when `i` <10 is false!). The second loop iterates j from

 © James Young, 2016

0...9. But how do they combine? Keep in mind that computers work on the mechanics. After the `i` loop is started, it runs the block. Inside the block, it starts the j loop. ***The catch is to realize that, each time the i-loop block is run, the whole 0...9 j loop runs.*** Let's step through

- count is set to 0
- the `i` for loop initiates, `i` is 0, block is run
  - the `j` loop initiates, `i` is 0, `j` is 0, block is run
    - count gets increased to 1.
  - the `j` loop checks `j<10`, true, `j` becomes 1, block is run
    - count gets increased to 2.
  - … continue until `j<10` is `false`
- the `i` loop checks `i<10`, true, `i` becomes 1, block is run again
  - the `j` loop initiates, `i` is 1, `j` is 0
  - the *entire* `j` loop runs again.
- … repeat while `i<10` is true

As this illustrates, the inside j loop gets completely run for each time through the outside block.

***Study the above steps. This is very important.*** One thing that I recommend for people to do is to try to solve a nested for loop on paper: first, make a chart of the current `i` and `j` values. Then, go through the loops step by step and record all the changes that happen to those variables.

What does print output? What is the final value of count? To figure this out, we do some basic math.

The `i` block runs 10 times. Each time the `i` block runs, the j block runs 10 times. Therefore, the j block runs 100 times total, and increments count 100 times. Count starts at 0, so count ends at 100.

To test this, try running the following code:

```
for (int i = 0; i < 10; i++)
{
  for (int j = 0; j < 10; j++)
  {
    println(i+" "+j);
  }
}
```

Here, the output shows you what happens to `i` and `j`. Is it what you expect? What changes if you reverse the order of the `for` loops?

 © James Young, 2016

13.4 **Example: A Dot Grid**

Use a nested loop to draw a grid of ellipses, 10 by 10. First, we make one `for` loop to go through all the columns.

```
for (int i = 0; i < gridSize; i++) // go through the columns
{
}
```

Let's also make a loop to go through all the rows. Since we will be combining these, let's use a different variable name

```
for (int j = 0; j < gridSize; j++) // go through the rows
{
}
```

Now we can combine these loops. The way we should think about it is as follows.

For each column in our grid…

Go through each row in the grid…

Draw an ellipse

Turning this into code:

```
for (int i = 0; i < gridSize; i++) // go through the columns
{
    for (int j = 0; j < gridSize; j++) // go through the rows
    {
      // draw the ellipse
    }
}
```

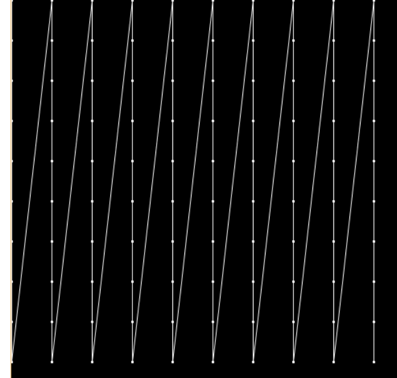
As in our previous example, the whole for loop on j gets run for *each value of* `i`. So, when `i = 0`, we will get j all the way from 0...9. Then, when `i =1`, we will get the whole range on j from 0...9 again. This way, we will hit every combination of `i` and `j` in the 0...9 range.

To draw the ellipse, we just need to convert from the grid coordinates to the screen coordinates. We have done this before. First we calculate how wide each grid cell is, and multiply it by how many grid cells along we are:

```
int x = width/gridSize * i; // column
int y = height/gridSize * j; // row
```

 © James Young, 2016

```
ellipse(x,y,2,2);
```



There is one final thing we can try. Even though we have a grid of 100 dots, and, we see the entire grid at once, the computer only draws one dot at a time. We can do a *trace* to see the order that the dots are drawn in. We can do this by drawing a line from the previous point, to the current point. That way, we can follow the line to see what order the points were drawn in. Try implementing this on your own, and then try changing the order of the `for` loops (`i` and `j`) to see what happens.

13.5 **Example: Raster**

"Raster" sounds complicated, but it's just the idea of setting each pixel color on the screen individually. This is how pictures work – it stores a brightness or color for each dot, and draws them on a grid, and you get a picture. Let's play with setting each dot's color based on some calculation to see how it looks.

To do this, we will use a nested for loop to go over every single pixel on the screen. Then, we will draw a point at each x and y coordinate. This is very similar to the previous example, except we can setup our loops based on the size of the window:

```
// go over each x coordinate
for (int x = 0; x < width; x++)
{
  // at each x, go over each y coordinate
  for (int y = 0; y < height; y++)
  {
    stroke(random(255));
    point(x,y);
  }
}
```

To recap – the first loop goes through each possible x coordinate across the screen. At each x coordinate (column), we then run down all the possible y coordinates. The whole y for loop is run at *each x*. This way, every pixel on the screen is touched. Here, we give it a random color, so we get what is called white noise.

If you do this as a static program you will quickly see the result. However, if you do it as an active program, you may notice that you program is very slow – if it's not, you have a fast computer! We are doing a lot of work here, and using some pretty

 © James Young, 2016

heavy-handed techniques. If you stick with computer graphics, you will learn advanced methods to speed this kind of operation up. For now here are two things you can do to make your program reasonably faster:

- Add the command `noSmooth();` to your setup block. This turns off some extra work the computer does to make your program look nice.
- Shrink your screen size. Half the size (250x250) actually has 1/4 the dots!

So far, this is a pretty boring example which sets every pixel to a random color. However, now that we have this basic setup, we can change things. How about we set the color based on how far it is from the mouse? Remember, the high-school math formula for calculating the distance between two points (in this case, the mouse and a point) is

$$\sqrt{(mouseX - x)^2 + (mouseY - y)^2}$$

We can do this easily. We already know how to subtract numbers. To square them, we can just multiply by itself, using some helper variables just for clarity. We just need one additional command, which actually came up in an exercise in Unit 7:

```
float sqrt(float number); // gives the square root of a number
```

And we can develop the following code:

```
float diffX = mouseX - x;
float diffY = mouseY - y;
float dist = sqrt(diffX*diffX+diffY*diffY);
```

Now that we have the distance, let's just set the point color to the distance:

```
stroke(dist);
```

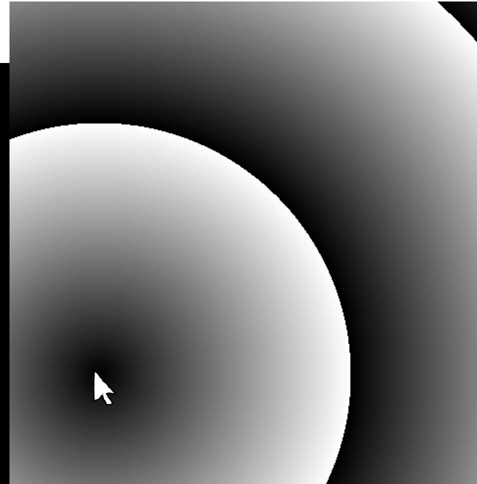
There is one small unsettling thing here, though. The distance on your screen between a point and the mouse may be larger than 255, our maximum color. Processing is very robust and doesn't mind you doing this, but we should be better behaved and handle it properly. Why don't we loop the color around, using modulo? That is, before drawing our point, let's instead calculate the color as follows:

```
float c = dist%256;
```

That is, if `dist` is bigger than 255, c will wrap around from 0 again. Your output should look like the inset on the next page. The hard line from white to black is where the distance equals 256 – this makes a perfect circle, which makes sense, as this defines the same distance from the mouse!

 © James Young, 2016

Here is my final program:



```
void setup()
{
 size(500,500);
 noSmooth();
}

void draw()
{
  background(0);
  stroke(255);
   for (int x = 0 ; x < width; x++)
   {
     for (int y = 0; y < height; y++)
     {
        float diffX = mouseX - x;
        float diffY = mouseY - y;
        float dist = sqrt(diffX*diffX+diffY*diffY);
        float c = dist%256;
        stroke(c);
        point(x,y);
     }
   }
}
```

You can play with what formula is used to set the pixel color, which can be a lot of fun. For example, try some of the following ones that I whipped up. Unless you are really mathematical, don't bother trying to figure out why they look the way they do

```
float c = (dist*dist)%256;
float c = (dist*x)%256;
float c = (dist+x-y)%256;
```

```
float c = (dist*x/(y+1))%256
```

Try to animate these visualizations by using a moving offset. That is, add an offset to the colors, and make the offset increase or decrease each frame. This will add a psychedelic effect!

13.6 **Example: Tic-Tac-Toe board**

Let's do a basic tic-tac-toe board. At this point we won't make a game of it, but will just put the basic board in place, and do hit detection with the mouse – to detect

 © James Young, 2016

which square is being clicked on.

The first part of this example is to draw the board. Since the tic-tac-toe board is only a 3x3, we could probably do it manually. However, it's cleanest if we do it using a nested for loop. The way to think about this is to use two loops – one for each dimension (rows, columns), and then realize that inside the loop, each cell of the game board is hit.

As always, we should make things as general as possible, so let's start by putting some global variables at the top of the program.

```
final int BOARD_GRID = 3; // how many squares
final int BOARD_SIZE = 250; // how many pixels
final int TILE_SIZE = BOARD_SIZE/BOARD_GRID;
int boardCenterX;
int boardCenterY;
int boardLeft;
int boardTop;
```

Why didn't I initialize the center coordinates, or the left and top coordinate? The reason is because these are dependent on the size of the canvas. Since we will change the canvas size in our setup code, we need to wait until the canvas is set before we calculate these:

```
void setup()
{
  size(500, 500);

  // center of the screen
  boardCenterX = width/2;
  boardCenterY = height/2;

  // board is centered, so left/top is half size off center
  boardLeft = boardCenterX - BOARD_SIZE/2;
  boardTop = boardCenterY - BOARD_SIZE/2;
}
```

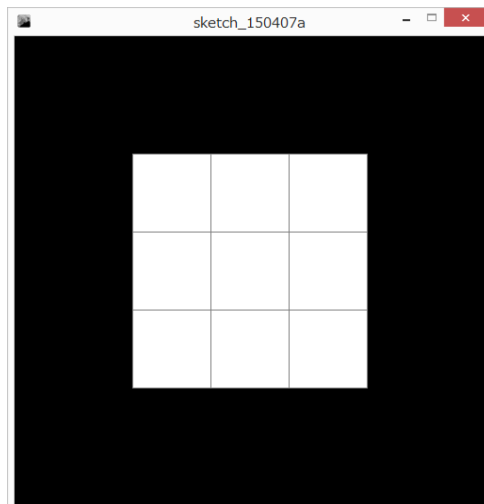
Now, we need to setup our `for` loop to draw the 9 tiles of our board. In this case, it makes most sense to use a nested for loop to hit each tile, with one loop going across the width of the board and the other across the height. First, let's setup our for loops:

 © James Young, 2016

```
for (int i = 0; i < BOARD_GRID; i++)
  {
    for (int j = 0; j < BOARD_GRID; j++)
    {
    // draw the tile i,j
  }
}
```

To draw the tiles of the board, we should use the `rect` command. This requires the top and left of the rectangle, as well as the width and height. We already know the width and the height, which we calculated as `TILE_SIZE`. The left and top can be calculated by starting at the left and top of the board, and coming in by how many tiles we are at. That is, the left of tile `i` is `boardLeft + i*TILE_SIZE`. The top is calculated similarly. Inside the nested loop, then we can do the following:

```
int left = boardLeft+ i*TILE_SIZE;
int top =  boardTop+ j*TILE_SIZE;
rect(left,top,TILE_SIZE,TILE_SIZE);
```



I also set the stroke color to 127 with the fill to 255. You should now see the image in the inset.

Now let's work on how to identify which box the mouse is in. Instead of thinking about how to do the math up front, the `for` loop already puts us within each square, one at a time, with the left and top calculated. We can do this right inside the loop and, when the mouse is inside a square, let's change the fill color to 127.

This is just like our previous example of clicking a button, except now we are doing it inside the `for` loop. I would add the following helper variables to make the whole thing clearer. Make sure to add these inside the nested loop since they are specific to the particular square:

```
int right = left+TILE_SIZE;
int bottom = top+TILE_SIZE;
```

Then, the logic to determine if the mouse is inside a square is straight forward. The only gotcha here is the borders. Pay attention to my particular solution and I'll explain below.

 © James Young, 2016

```
if (mouseX >= left && mouseX < right && // in X range
  mouseY >= top && mouseY < bottom) // in Y range
{
  fill(127); // a hit
}
else
{
  fill(255);
}
```

The logic is straight forward – if it's bigger than the `left` AND less than the `right`, we are in the X range. Also, if it's below the `top` AND above the `bottom`, then we must be inside the tile. In my case, I used the >= on one side and the < on the other side. The equals includes the border, so what I effectively did was to include the left and top border in this square, and exclude the bottom and right border (for the next squares). Quickly add additional logic to test if the mouse is pressed as well. You now have a basic setup for a tic-tac-toe board! Here is my final code:

```
final int BOARD_GRID = 3; // how many squares
final int BOARD_SIZE = 250; // how many pixels
final int TILE_SIZE = BOARD_SIZE/BOARD_GRID;
int boardCenterX;
int boardCenterY;
int boardLeft;
int boardTop;

void setup()
{
  size(500, 500);

  // center of the screen
  boardCenterX = width/2;
  boardCenterY = height/2;

  // board is centered, so left/top is half size off center
  boardLeft = boardCenterX - BOARD_SIZE/2;
  boardTop = boardCenterY - BOARD_SIZE/2;
}

void draw(){
```

 © James Young, 2016

```
  background(0);
  stroke(127);
  for (int i = 0; i < BOARD_GRID; i++)
  {
    for (int j = 0; j < BOARD_GRID; j++)
    {
      // borders of tile i,j
      int left = boardLeft+ i*TILE_SIZE;
      int top =  boardTop+ j*TILE_SIZE;
      int right = left+TILE_SIZE;
      int bottom = top+TILE_SIZE;

      if (mouseX >= left && mouseX < right &&
          mouseY >= top && mouseY < bottom &&
          mousePressed)
      {
          fill(127);
      }
      else
      {
         fill(255);
      }

      rect(left,top,TILE_SIZE,TILE_SIZE);
    }
  }
}
```

There are a few things to note here. If you hold the mouse button down and drag it across the board, the tile that is highlighted changes and follows the mouse, where you would want only the tile that was actually clicked to change. You can achieve this with some clever logic that you would have seen earlier, in the boolean logic chapters. Another way is to use advanced event handling (look up the `mousePressed()` method function on the reference manual) that we will not learn quite yet.

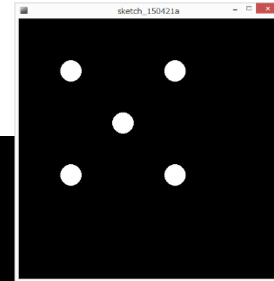
Also, you may want to save some game state into variables. You would need a variable for each square, and although this is possible with the tools you learned so far, it would be very tedious.

Try updating the board to draw an X or an O in the square when clicked!

 © James Young, 2016

13.7 Example: draw a dice face

While this may not be the simplest way to do it, this technique for drawing a dice face is great practice for your nested for loops and boolean logic. First, let's setup a nested for loop that draws a 3x3 grid of dots. Here are my supporting globals:




```
int DICE_GRID = 3; // 3x3 dice face
int DICE_SPACING = 100; // space between dots
int DICE_LEFT = 100; // dice position
int DICE_TOP = 100;
int DOT_SIZE = 40;
```

Then, the for loop setup is straight forward from examples we did previously:

```
// iterate through the columns
for (int i = 0; i < DICE_GRID; i++)
{
  for (int j = 0; j < DICE_GRID; j++) // rows
  {
    // calculate position of dot
    int x = DICE_LEFT+i*DICE_SPACING;
    int y = DICE_TOP+j*DICE_SPACING;

    // draw dot
    ellipse(x, y, DICE_TOP, DICE_TOP);
  }
}
```

This will give you a 3 x 3 grid of dots. To make a dice shape, we need to put an `if` statement in front of the `ellipse`, to only draw dots when certain conditions are met. Let's think about the dice #3, first, which consists only of a diagonal. Well, how do we define a diagonal in our grid? One diagonal is when `i = j`: `0,0 1,1` and `2,2`. So let's try changing the ellipse code as follows:

```
if (i == j)
{
  ellipse(x,y,dotSize,dotSize);
}
```

You now should have a 3 dice! Let's do 5 next, which is a little trickier. First, we can see that we can make a 5 by doing two diagonals. We have the one diagonal, but how can we get the other? Try writing the coordinates out:

 © James Young, 2016

2,0 1,1 0,2. Can you see a pattern? This is similar to the diagonal one above except the x coordinate is backwards. As i goes up, 0,1,2, how can we convert that i into 2,1,0? You can take the maximum, 2, and subtract i from it. This is confusing so check out the following table

i	2-i	j
0	2	2
1	1	1
2	0	0

Still confused? Draw it out on paper. Notice how I put j down backwards? This illustrates that when 2-i == j, then our i,j gives us the coordinates we want. So, we can get the coordinates for the opposite diagonal when (2-i) == j.

```
if ( (2-i) == j)
{
  ellipse(x,y,DOT_SIZE,DOT_SIZE);
}
```

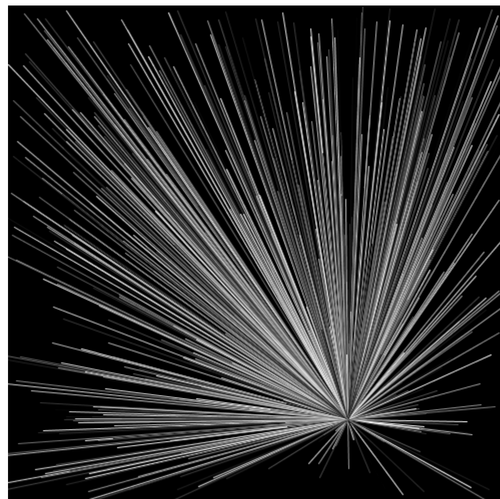
Finally, we can combine the two diagonals with a simple OR operator:

```
if ( i==j || (2-i) == j)
{
  ellipse(x,y,DOT_SIZE,DOT_SIZE);
}
```

And we're done!

13.8 **Example: Nested While Loop**

While loops can be nested just like other loops, if statements, etc. Let's do a quick example.



We will make a program that acts unpredictably. The interface pauses for a random amount of time, and draws lines to the mouse point – the longer it paused, the more lines you get.

There are a lot of ways to make a program pause. We will use a technique called a **busy loop**, which basically makes work and wastes computer power to kill time. In a later course you will learn why this should be avoided at all costs,

 © James Young, 2016

but it's a fun exercise ☺.

In order to wait a random amount of time, we will basically roll dice.

- Pick a random number up to `MAX`
- If the roll is a 0, we're done. While it's NOT zero,
  - Pick another random number.

The larger that `MAX` is, the less likely we are to get a 0, and the more times the loop runs. Next, let's add some lines EACH TIME we don't roll a zero. Our updated algorithm is:

- Pick a random number up to `MAX`
- If the roll is a 0, we're done. While it's NOT zero,
  - Pick another random number.
  - Draw 5 lines from random locations to the mouse.

Keep in mind that while this loop is running, the draw block never finishes, and the screen doesn't update. So, the longer the while loop runs, the longer the animation freezes.

Implement a `while` loop for the outer random number guesser first. Approach this with the priming, test, and upkeep mentality. I have `ODDS` set to 500.

```
int i = (int)(random(ODDS));
while (i != 0)
{
  // draw lines
  i = (int)(random(ODDS));
}
```

We start with a random number. If it's not 0, we pick another one, and check again. Repeat. This will take some time.

Now we need to update the loop, and each time we get a random number, we need to draw that many lines. The nested loop we could do with a `for` loop, but use a while for practice. This should be straight forward, so try it on your own.

Check your Understanding

13.9 Check Your Understanding: Exercises

Exercise 1. Re-implement the dot grid example (draw a 2D grid of dots), and make the following changes:

 © James Young, 2016

a. Make the size of the ellipse depend on how far it is from the mouse. Figure out the maximum distance, and linearly scale the actual distance to a reasonable size range.
b. Make the color of the ellipse depend on how far it is from the mouse. Use the same logic as in a, but map to colors instead.

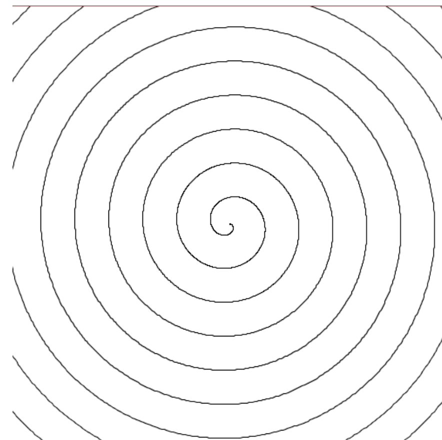
Exercise 2. Update the dice face example to include all the usual 6 numbers. In each case, only use one if statement, and try to minimize how many checks you need. 4 may be the hardest, as you can do it with just two equal's tests.

a. Try implementing functions to draw the faces and an if-else chain to turn a number into a graphic. You can now use this in a game.

Exercise 3. Create a nested for loop to iterate over each pixel on the screen, after first clearing the screen to black. Then, use the following logic to choose the color:

a. If x*y is divisible by 2, then make the color white. Otherwise, make the color black. How does it look?
b. Change 2 to 3, then 4, etc., and run the program each time. Do you see a pattern? What numbers give you a grid of perfect squares? Why would this be? (it's tricky).

Exercise 4. Computers might help you hypnotize your friends! You will make a hypnosis-helper tool, as shown in the screenshot at right. In the Exercise 4 in the previous Unit, you drew an approximated circle by using angles that increased in a `for` loop. To make a spiral instead of a circle, increase the radius, too, as you go along. Then the spiral can easily be animated in several different ways. Implement this program in three short phases:



a. Draw a spiral using a single for loop, and basic trigonometry. First, define the following named constants:

   `DRAW_STEPS`: The number of line segments that you will draw to make the spiral. Try a value of 250 to start with. This must be an integer variable, so that you can reliably use it in a for loop.

   `MAX_RADIUS`: The maximum radius that will be reached when the last line is drawn. Use a 500 by 500 window, and set this value to $250\sqrt{2}$. (That's the distance from the centre to any corner. If you change the window size, you'll have to change this value, too.)

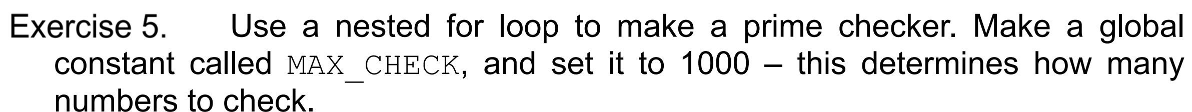
 © James Young, 2016

`numTurns`: the number of turns you want in your spiral. This is not a named constant because we will change it later. Try a value of 5 to start with.

Now use a single for loop to draw exactly `NUM_STEPS` lines. The first one should start in the center of the window. Use the command `noSmooth();` in the `setup()` block to speed up this program if it's slow. Each line should connect the previous point to a new point (remember the previous point). The first point is the window's center.

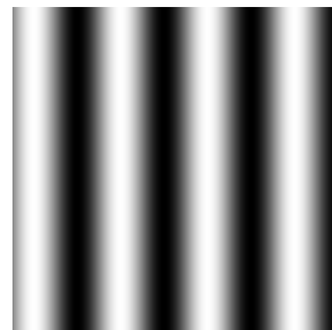
The angle to the new point on the spiral should be 0 for the first line, and increase steadily, by equal steps, to 2π*`numTurns` for the last line. You will need to do linear scaling from your steps to the radians for this.

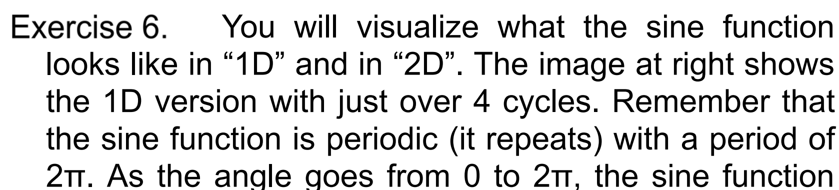
b. Make the spiral rotate by adding a constant "offset" value to the angle (it no longer goes from `0` to `2π*numTurns`, but instead goes from `offset` to `2π*numTurns+offset`). Start this `offset` at 0, and increase it slightly each time you draw. Try increasing it by 0.1 radians each time (use a named constant, of course). The spiral should now rotate. ("You are getting sleepy.")
c. Also make `numTurns` increase slightly every frame. Try making it change by 0.1 turns per frame (use another global named constant). More and more "loops" of the spiral should be drawn as this value increases. They will lose their smoothness, since more turns are being drawn, but the number of lines used to draw them hasn't changed. This produces a cool effect. Now change your program so that the turns of the spiral stay equally smooth as the number of turns grows. (This isn't as cool, though, so save the old code, too.)



Exercise 5. Use a nested for loop to make a prime checker. Make a global constant called `MAX_CHECK`, and set it to 1000 – this determines how many numbers to check.

a. Make a loop to go from 1 to `MAX_CHECK`, inclusive
b. For each of these numbers, use a second loop to count how many factors it has. Hint: a number `n` is a factor of `p` if it divides evenly into p. (use mod). If the factors are exactly 2, then print out a statement telling the user that the number is a prime.
c. If you increase `MAX_CHECK`, this quickly gets very slow. Can you think of an easy way to speed this up?

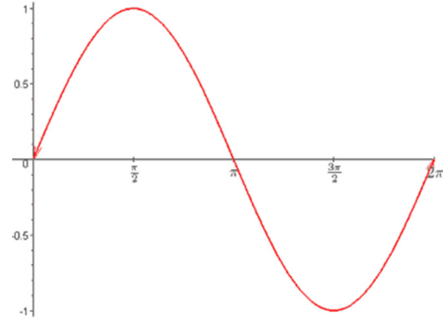




Exercise 6. You will visualize what the sine function looks like in "1D" and in "2D". The image at right shows the 1D version with just over 4 cycles. Remember that the sine function is periodic (it repeats) with a period of 2π. As the angle goes from 0 to 2π, the sine function

 © James Young, 2016

goes from 0 to +1, then to -1, then back to 0. You will map a sine value (-1 to +1) to 0...255 to obtain a greyscale color. -1 should give black, 0 should give medium grey, and 1 should give white. That is what you see in the image above.



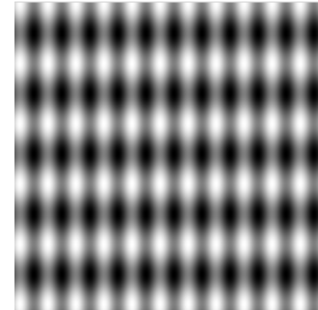
a. Use a for loop and tightly-spaced vertical lines (one per x coordinate across the screen), to draw an image such as the one shown on the previous page. (add `noSmooth();` to `setup()` to increase speed and use a smaller canvas size, such as 300 by 300).

   Set a global variable `numCycles` (in part b it will change, so not a constant) to the number of cycles of the sine function to plot (try 4.0).

   Set the line color based on the x coordinate, by converting the position from `0` to `width-1` to the sine angle, `0 to numCycles*2*PI` (hint: use percentages). Then translate the output from the sine of this value (-1 to +1) to a greyscale value (0 to 255).

b. Now you can move to an animated 1D image. Make `numCycles` controlled by `mouseX` . It should change from `0` to `MAX_CYCLES` as `mouseX` changes from `0` to `width-1`. Try `MAX_CYCLES` at 12.

c. Now, we will move to a 2D image. Instead of using lines, you will need to set the color at each pixel using the `point` command. Use two for loops, nested, to hit every pixel. First, simulate question b by setting the color based on the x only, as above. You should get the same output (but probably slower).



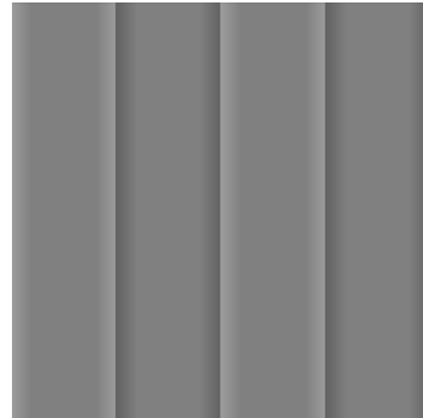
   Now, update your program to also do a sine calculation in the y direction. You will need to set the number of cycles in the y direction, and repeat all the sine and scaling calculations to get a y value. First, ignore the x and set the color based on the y only. Once this works, then average the x and y color to get your result. This is what the sine function looks like in 2D!

   Try using color!! Instead of averaging the colors, try using `stroke(xColor, yColor, 0);` which looks cool!

Exercise 7. Optical illusions!! Quick! Are the alternating bands in the image below the same color, or are some lighter or darker than the others? They are actually exactly the same color, with the only color differences being at the borders! This is called the Cornsweet effect. You will implement it so that you know for yourself that the bands are of the same color.

 © James Young, 2016

a. Basic image. Use a global constant `BANDS` to control the number of vertical bands in the image. The image at right uses 4 bands. Most of each band will be the same color (128, medium grey), but the bands should either darken or lighten at the edges. Use global constants to control what proportion of each band is the “fade in/out” region on each side, and how much the color will change from the default value of 128. (The image shown uses 20% on each side – 40% total, and fades in/out by 100 – giving grey values from 28 to 228.)



   Use for loop(s) to draw the image. You can either use one loop to generate every x coordinate across the canvas, but it may be easier to use two: one for the band number, and one for the position within the band. You can do either one. Use vertical lines, not individual points, to draw the image, spaced tightly so that you draw them 1 pixel apart (or very close to it).

   Even numbered bands (if you start at band 0) should brighten from `128` to `128+MAX_CHANGE` on both edges, but odd numbered bands should darken from `128` to `128-MAX_CHANGE` instead. Notice that the change is smooth, with a linear change.

b. Animated image. Link the amount of fade to the `mouseX` value, so that when the mouse is at the left (`0`), there is no fade out at all (so you should get a solid grey screen), and when the mouse is at the right (`width-1`), the bars change completely to white/black at the edges. Now you can see the illusion more clearly.

Exercise 8. You will draw a message, with each character having its own size and color, as shown on the right, with the size proportional to the frequency of the letter in the message. This is not an Active Processing program (there is no animation).

a. Store the message in a global constant, and only use lower-case letters. Determine algorithmically which character appears the most in the text. That is, find the maximum frequency of occurrence of any character in `MESSAGE`. Do not count vowels (a, e, i, o, u) or blanks, since they occur far too often, and spoil the effect. Print the character with the maximum frequency, with its frequency (count), to the console, to test this. In

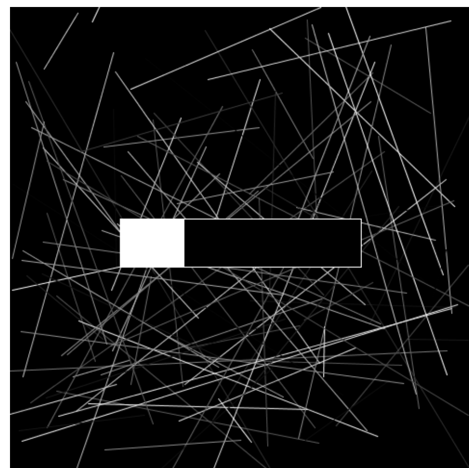
 © James Young, 2016

the sample shown, 's' appears most often (7 times), whereas 'z', 'b', and '!' only appear once.

b. Go through the `MESSAGE`, drawing one character at a time to the canvas. Each character should have its own size and color. Vowels (a, e, i, o, u) should be fixed to `VOWEL_COLOR`, and all other characters should be `TEXT_COLOR`, drawn against a `BG_COLOR` background. A non-vowel, non-blank character having the maximum frequency (determined in part a) should be `LARGEST_TEXT` pixels in size, whereas one that only appears once should be `SMALLEST_TEXT` pixels in size. Others should have a size somewhere in between, proportional to their frequency. All vowels and blanks should have a size halfway between those two extremes, regardless of frequency. To space the characters, you can use the function `textWidth(s)` to find the width `w` of any string `s`, in pixels, at the currently selected font and size. (Because anything can be converted to a String, s can really be any data type at all.) Begin the next character w pixels to the right of the current character, where w is the width of the current character. To test the program so far, draw characters starting at x=0, y=height/2, and let them disappear off the right edge of the canvas.

Phase 3: Draw the message on as many lines as necessary. Draw all lines starting at x=MARGIN. Draw the first line with the characters at y=FIRST_LINE_Y. Whenever you have to start a new line, increase the y coordinate by LINE_SPACING. Draw as many characters as you can on the current line. But if drawing the current character would go beyond width-MARGIN, then start a new line before drawing that character. Now the program should give the result shown in the sample image.

Exercise 9. Make a status bar on a prime number calculator. When computers are working hard, users want feedback on the progress. If the computer waits until it is done all the work before giving an update, the user may be waiting a very long time and may worry that the program crashed. Status bars are a great way of keeping users informed.



You will implement a program that finds the 10000th prime number. Here are my globals: `PRIMES_TARGET = 10000`, `BATCH_SIZE = 100`, `STATUS_BAR_WIDTH = 250`, `STATUS_BAR_HEIGHT = 50`, `primesFound = 1;` // assume that 1 is already counted, `nextPrime = 2`, `done = false`;

We defined a `BATCH_SIZE` – this tells us how many numbers to test if they are prime, before updating the user. Here is a sketch of the program:

 © James Young, 2016

- Do a batch (use a for loop) of numbers to test, run the loop `BATCH_SIZE` times
  - Check if `nextPrime` is a prime. See exercise 5 above:
  - Increase `nextPrime` for the next check.
  - if we found enough primes, change our boolean to signal we are done. And don't do any more simulations. NOTE: what if you are in the middle of a batch?? How do you end the for loop early? Hint: modify the for loop condition to also rely on the `done` variable
  - draw a random line with a random color to signify that a number was checked
- After the batch is done, draw the status bar using two rectangles, which shows how many primes were found out of our target maximum. Hint: use two rectangles, one for the whole box, one for the filled amount.

This example illustrates an important usability issue. It is important to give people feedback, but the computer has to do wok to provide this feedback. If you provide too much feedback, it slows the real work (prime checking) down. Play with the batch size to see how the speed changes. Also note that the checking is slower as you go along, because it takes more work to check a large number than a small one.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

 © James Young, 2016

(page intentionally left blank)

 © James Young, 2016