The while Loop Again :
The Processing API explains the while loop as shown below:

Name	**while**

Examples

```
int i=0;
while(i<80) {
  line(30, i, 80, i);
  i = i + 5;
}
```

Description

Controls a sequence of repetitions. The **while** structure executes a series of statements continuously while the **expression** is **true**. The expression must be updated during the repetitions or the program will never "break out" of **while**.

This function can be dangerous because the code inside the while() loop will not finish until the expression inside while() becomes true. It will lock out all other code from running (mouse events will not be updated, etc.) So be careful because this can lock up your code (and sometimes even the Processing environment itself) if used incorrectly.

Syntax

```
while (expression) {
  statements
}
```

Parameters		
	expression	a valid expression
	statements	one or more statements

The expression must be a boolean expression. Remember that boolean expressions evaluate to either true or false.   Your code may use any of the following As the boolean expression:

- an expression that uses the relational operators:
  == < >   <=   >= !=
  and possibly the logical operators
  &&   ==   !

- a boolean variable

- a function that returns a boolean value

The for Loop again:
The Processing API explains the for loop as shown below:

Name   for

Examples

```
for (int i = 0; i < 40; i = i+1) {
    line(30, i, 80, i);
}
```

```
for (int i = 0; i < 80; i = i+5) {
    line(30, i, 80, i);
}
```

Description   Controls a sequence of repetitions. A **for** structure has three parts: **init, test,** and **update.**
Each part must be separated by a semi-colon ";". The loop continues until the test evaluates to
false. When a **for** structure is executed, the following sequence of events occurs:
1. The init statement is executed
2. The test is evaluated to be true or false
3. If the test is true, jump to step 4. If the test is False, jump to step 6
4. Execute the statements within the block
5. Execute the update statement and jump to step 2
6. Exit the loop.

Syntax
```
for (init; test; update) {
    statements
}
```

Parameters   init   statement executed once when beginning loop

test   if the test evaluates to true, the statements execute

update   executes at the end of each iteration

statements   collection of statements executed each time through the loop

We usually use a for loop when we know exactly now many times we want to
repeat some action or set of actions. It turns out that the for loop is ideal for our
last major new area of work – the dreaded, evil, terribly complex, guaranteed to
fail you topic of   **ARRAYS . . .**

And now – the Array:

Arrays are not that evil or terribly complex. You have existed with a set of arrays since you were born. You have intelligently worked with these arrays ever since you can remember thinking about anything. Any component of your life that consists of a sequence of numbered events is an array:

- hours in a day
- days in a week
- years
- cents in a dollar
- semesters in a college career

All of these are arrays. Some are numbered nicely like the years or the days of the month. Others use words to represent numbers such as Monday or January. We understand that Monday is the second day of the week and first day of the work week.

Some of these arrays are multiple dimensioned arrays. Consider years:

- one dimension is the sequence of years
- within each year is a second dimension -- months
- within each month is a third dimension – days
- within each day is a fourth dimension – hours
- . . .

Most buildings are two-dimensional arrays:

- floors are numbered 1 - ??
- rooms are numbered starting with the floor number[1]

Arrays are very useful for storing large amounts of similar (meaning the same type of) data. Jim ran a program in class that showed you only a little bit of code:

```
// Arrays Opening Demo
// change this number to change the number of squares.
final int MAX = 1;
```

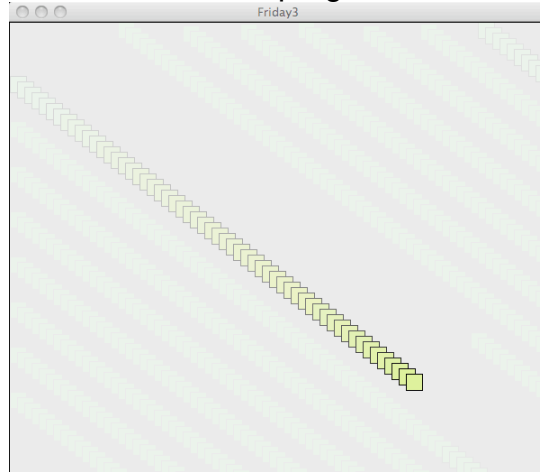This line declares a constant – not a variable. It is a constant because of the first word on the line:

<p align="center">final int MAX = 1;</p>

The word final means that 1is the only or final value that MAX will ever have. The convention among many programmers is that constants have names that are all upper case letters. If we remove the word final, then max becomes an ordinary variable.

---

[1] Except for Gates – Jim's office used to be 6019. There was no office beneath him and the office above him was 7021??? The reason there was no office beneath his office was because his office was literally stuck on the side to Gates and suspended out in thin air (aided by several terribly thin columns of concrete – not that this bothered him in any way… *really*. . .
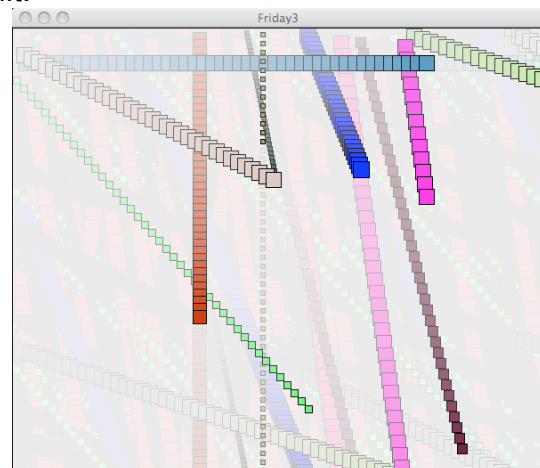
Here is the result of the execution of the program that has this line of code:

Then Jim changed the value of MAX to 10:

<p style="text-align:center;">final int MAX = 10;</p>
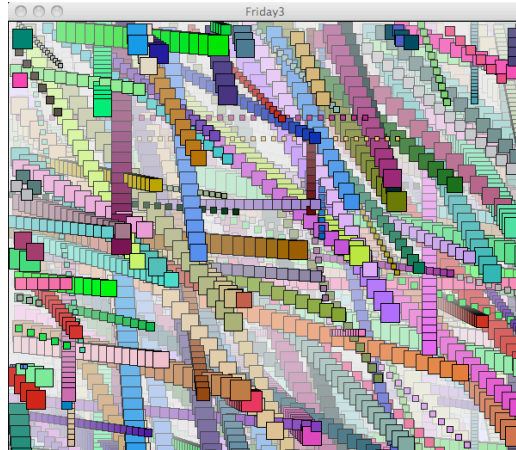
and this was the result:

A single edit produced this effect. How many lines of code would it take you to alter the program that draws the first screen to produce the output in the second screen? More than 1???

Here was the second change Jim made in class:

<p style="text-align:center;">final int MAX = 100;</p>

which produced this output:

How many edits would it take to you produce this? It would take a lot of edits. The difference between the single edit that Jim made and the edits you would have to make is the result of one idea – the array. If we want to animate something, we will probably have more than a few values to animate. Using individually named variables for all of those values will work but it will become very difficult to manage and very time consuming to change. The array offers us a way to increase (or decrease) the amount of data in our programs is a very manageable way.

And so we begin our last new MAJOR area of work.

Here is Processing's API entry for arrays:

Name

Array

Examples

```
int[] numbers = new int[3];
numbers[0] = 90;
numbers[1] = 150;
numbers[2] = 30;
int a = numbers[0] + numbers[1]; // Sets variable a to 240
int b = numbers[1] + numbers[2]; // Sets variable b to 180
```

```
int[] numbers = { 90, 150, 30 };
int a = numbers[0] + numbers[1]; // Sets variable a to 240
int b = numbers[1] + numbers[2]; // Sets variable b to 180
```

=>   <=

```
int degrees = 360;
float[] cos_vals = new float[degrees];
for(int i=0; i < degrees; i++) {
  cos_vals[i] = cos(TWO_PI/degrees * i);
}
```

Description

An array is a list of data. It is possible to have an array of any type of data. Each piece of data in an array is identified by an index number representing its position in the array. The first element in the array is **[0]**, the second element is **[1]**, and so on. Arrays are similar to objects, so they must be created with the keyword **new**. Every array has a variable **length** which is an integer value for the total number of elements in the array. (People are often confused by the use of **length()** to get the size of a String and **length** to get the size of an array. Notice the absence of the parentheses when working with arrays.)

This shows three different ways to build arrays. We will use the middle example for most of our work. Later we will use the first and third examples.

The syntax for the array is the set of [ ]² which means we have an array. Arrays are like variables in that they have a <u>name</u> and a <u>type</u>. The difference is that they can have more than a single value. This syntax:

<div align="center">int number = 42;</div>

declares a variable named number configured to store int values and it has a value of 42. It can only have one value. If we execute this syntax:

<div align="center">number = 2;</div>

we destroy the value 42. The variable number can only store one value at a time.

If we alter the syntax slightly:

<div align="center">int [ ] numbers = { 42. 2 };</div>

we can store multiple values.

The [ ] after the int converts the variable from a primitive (can only store one value) to an array.   We use the word "reference" to describe the array. The idea of a reference is somewhat subtle. Its meaning will become apparent later in our work with arrays. One way to think of the idea of a reference is to consider the number of our classroom.   5202 Wean is a reference to a space in a building. There is nothing inherent in the room that makes it 5202. The room number 5202 is just a reference to a space. If we all use this same reference, we get to the same place. The room we call 5202 Wean has other references that we as members of the class understand when we are talking to each other:
-   "I will meet you at the classroom"
-   "I left this in Jim's classroom"
-   "My programming classroom has a weird setup"

It turns out that one array can have multiple references while primitive variables can have only one name. Again, we will look at this idea a bit later.

The stuff in blue on the right side of the assignment operator:

<div align="center">int [ ] numbers = { 42. 2 };</div>

is called the initializer list. This initializer list tells Processing to allocate enough memory space to store two int values and to assign the values 42 and 2 to that space. The array has permanent, unchangeable space for 2 int values. This cannot shrink to 1 or zero or grow to 3 or larger.   Processing automatically attaches one additional piece of information to the array: its length. The length is a constant and is often called a field. The value of length is the number of values we 'can store in the array. In this case, the value of length is 2. To access this field, we use the dot or period notation in a manner similar to the width or height of an image:

<div align="center">println(numbers.length);</div>

If we want to use (access) one of the values in the array, we have to tell processing which value. We do this using the [ ] like this:

---

² These are brackets or, to may programmers, square brackets

println(numbers[0]);

This will print the value 42. This syntax:

println(numbers[1]);

will print the value 2;

OK something strange is going on here. This is a deeply and closely held secret by the cult of folks known as "programmers". The secret is that when we are working with arrays, we start counting at zero instead of 1.

The numbers in the brackets:

println(numbers[0]);
println(numbers[1]);

are called indexes or indices. The indexes are the locations of the values stored in the array.

Each value stored in the array is called an element of the array.

The array numbers has a length of 2 so it has 2 elements. The beginning element has an index of [0]. The last element of the array numbers has an index of [1].   This code:

println(numbers[0]);

will produce this output:
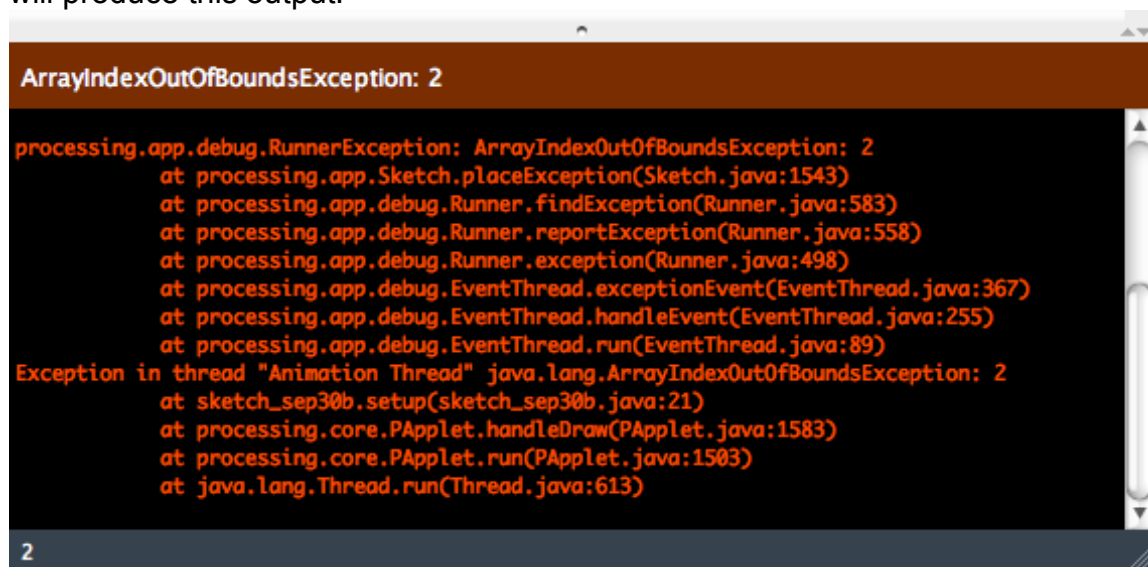
```
42
```

This code:

println(numbers[1]);

will produce this output:

```
2
```

This code:

println(numbers[2]);

will produce this output:

```
ArrayIndexOutOfBoundsException: 2

processing.app.debug.RunnerException: ArrayIndexOutOfBoundsException: 2
        at processing.app.Sketch.placeException(Sketch.java:1543)
        at processing.app.debug.Runner.findException(Runner.java:583)
        at processing.app.debug.Runner.reportException(Runner.java:558)
        at processing.app.debug.Runner.exception(Runner.java:498)
        at processing.app.debug.EventThread.exceptionEvent(EventThread.java:367)
        at processing.app.debug.EventThread.handleEvent(EventThread.java:255)
        at processing.app.debug.EventThread.run(EventThread.java:89)
Exception in thread "Animation Thread" java.lang.ArrayIndexOutOfBoundsException: 2
        at sketch_sep30b.setup(sketch_sep30b.java:21)
        at processing.core.PApplet.handleDraw(PApplet.java:1583)
        at processing.core.PApplet.run(PApplet.java:1503)
        at java.lang.Thread.run(Thread.java:613)

2
```

The array has two places to store int values. The indexes of those two places are [0] and [1]. The index [0] is called the lower bound of the array. The index [1] is the upper bound of the array. There is no index [2] so [2] is said to be "out of bounds". If we attempt to use an index of an array that does not exits, Processing throws a fit. This fit is called an exception. Processing is said to "throw an exception." It turns out that there are many fits… er.. exceptions that Processing can throw when it runs our programs. This is the first of many we will see in the coming weeks.
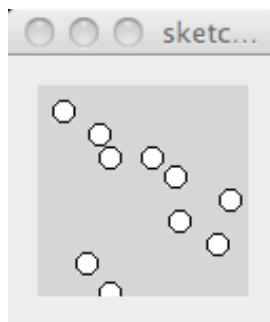
An array of two elements may not be useful. What about 10 **elements**?

```
int [ ] xValues = { 34, 65, 12, 85, 34, 29, 91, 23, 54,  67 };
int[ ] yValues = {  98, 43, 12, 75, 34, 23, 54, 84, 34, 64 };
```

These two arrays can be use to draw a series of circles or rectangles. To use these we can take advantage of the **for** *loop. We use the* **for** *loop because we know exactly how may iterations we need. We need 10 because both arrays have 10 elements. This code:*

```
for( int i = 0; i < xValues.length; i++)
{
    ellipse( xValues[i], yValues[i], 10, 10 );
}
```
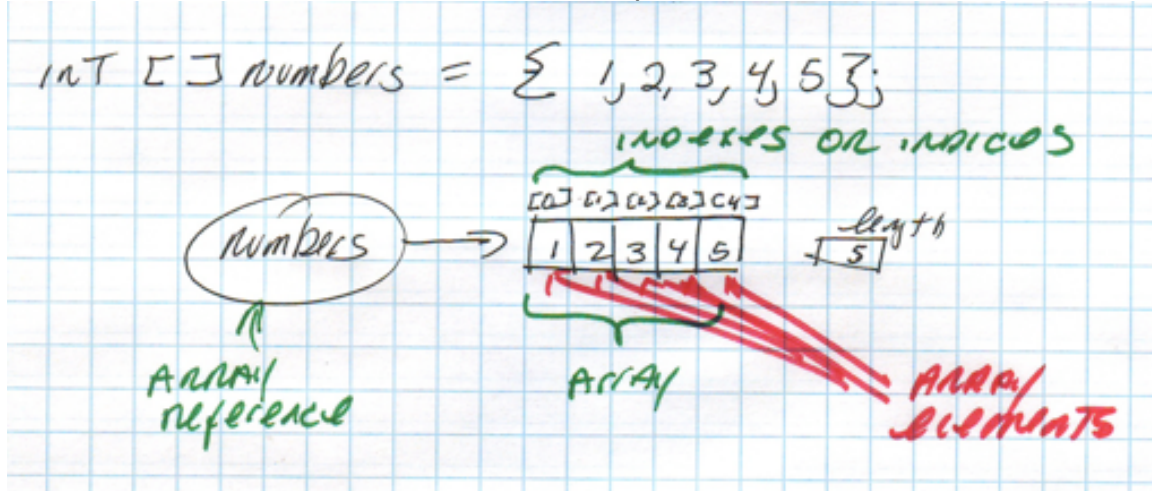
gives us this output:

The for loop repeats ten times. Each iteration changes the value of the variable, i starting with the value 0. When the value of i is 0, Processing uses the [0] element of the xValues array as the x coordinate of the circle and the [0] element of the yValue array as the y coordinate to draw the first circle. For the second iteration, the value of i is 1. Processing uses the [1] element of the xValues array as the x coordinate of the circle and the [1] element of the yValue array as the y coordinate to draw the next circle. This continues for elements 3, 4, 5, 6, 7, 8, and 9. When the value of i is 10, the iteration stops.

These two arrays are called "parallel arrays." They are parallel because elements with the same index are used to draw one ci rcle.

As we work with arrays, we need a uniform way to represent them graphically and we need a uniform set of names for the parts. Here is how we will do this:

Note that this jargon is just a important as the jargon used to identify parts of functions and will be tested in a similar manner.

The class code for today demonstrates some work with arrays. You should look at this carefully for the next class. Also, Shiffman does a very nice job of explaining arrays.

And remember, this is the first pass – we will revisit a lot of this over and over and over and over and over and over and over and over and over