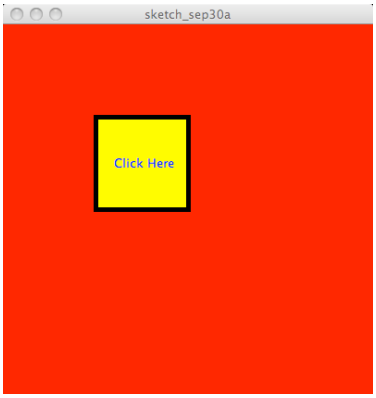
The && Operator (The AND operator):

This refers to the class code Set 10 .

Our use of the **if/else** *structure has been helpful. However, some of you have had to use it in a very difficult manner. This was because you had not yet been introduced to some syntax that would have made your coding easier.*

If you were trying to see if a click was within a rectangle, the **dist( )** *function was not really useful.*

Suppose we have a program that has a single button:



and that the location of the button is determined by the values of these four variables:

```
int buttonLeftX, buttonRightX, buttonTopY, buttonBottomY;
```

Here is the code that we would have to write to see if the user clicked ***inside*** *the button:*

```
if ( mouseX >= buttonLeftX )
{
   if ( mouseX <= buttonRightX)
   {
      if ( mouseY >= buttonTopY)
      {
         if ( mouseY <= buttonBottomY )
         {
            println("Button Clicked");
         }
         else
         {
            println("Button Missed");
         }
      }
      else
      {
         println("Button Missed");
      }
   }
   else
   {
      println("Button Missed");
   }
}
```

This code works properly but it is structurally very ugly. It is easy to misplace a brace and horrible to debug if it is wrong. Some of you actually have written code like this.

There is a better way - we can use the logical **AND** *operator which is:*

&&

This is two **ampersand***s or what most people call the* **AND** *sign.*

The logical **AND** *in Processing works just like it did when you studied it in your earlier math classes prior to arriving on the shores of CMULand…(when you were young, healthy, carefree, well fed, … Sorry – I digress…)*

Using the **&&** *operator, this code can be written like this:*

```
if ( mouseX >= buttonLeftX  &&   mouseX <= buttonRightX &&
     mouseY >= buttonTopY   &&   mouseY <= buttonBottomY )
{
    println("Button Clicked");
}
else
{
    println("Button Missed");
}
```

Everything within the parentheses of the **if** *is a single* **boolean** *expression. It is composed of* **boolean** *sub-expressions that are separated by the* **&&** *operator. The entire expression evaluates to* **true** *ONLY if all of the sub-expressions are* **true**. *If any one (or more) of the sub-expressions is* **false**, *the entire expression is* **false**.

When the **&&** *operator is used, everything must be* **true** *for the expression to evaluate to* **true**. *Any single evaluation to* **false** *makes the entire expression* **false**. *It is ,* "**all or nothing.**"

This syntax is much easier to write, read, debug, and understand.

There is a Logical **OR** *operator. It is:*

||

This operator is composed of two characters called **pipes**. *On Jim's keyboard (a Mac), this key is the shifted character on the backslash key under the delete key.*

Supposed we reverse the test we made in the previous code. In the previous code we tested to see if the user clicked inside the button. Let's write it to see if the user missed the button.

```
if ( mouseX < buttonLeftX  ||  mouseX > buttonRightX ||
     mouseY < buttonTopY   ||  mouseY > buttonBottomY )
{
    println("Button Missed");
}
else
{
    println("Button Clicked");
}
```

The expression with the **||** *operator evaluates to* **true** *if any one or more of the sub-expressions is* **true**. *All of the sub-expressions must be* **false** *before the entire expression evaluates to* **false**. *In this code each sub-expression tests to see if the click is outside of one side of the button's boundaries. The only way this entire expression can be false is if the user clicks inside the button's boundaries.*

Using either the **&&** *operator or the* **||** *operator can simplify your code. However, mixing them together can be a nightmare because of precedence rules. You can avoid this by using parentheses to force the order of evaluation that you want Processing to follow. Coding with the* **&&** *and/or the* **||** *operators is usually simpler if you write one sub-expression at a time and test each one as you go. Attempting to write the entire expression before compiling and testing can be a difficult way to write you code.*

Let's look at a ***verbose version*** *of the code Jim used in class to demonstrate this.*

```
void mousePressed( )
{
   if (mouseY >= height*.89) // #1
   {
      if ( mouseX >= 0 && mouseX <= width*.25) // #2
      {
          tint( 255, 0, 0 );
      }
      else  if ( mouseX >= width*.26 && mouseX <= width*.5)  // #3
      {
          tint( 0, 255, 0 );
      }
      else  if ( mouseX >= width*.51 && mouseX <= width*.75) // #4
      {
          tint( 0, 0, 255 );
      }
      else      // #5
      {
          tint( 255 );
      }
   }
}
```

#1 This is an outer **if**. *It is testing to see if the user clicked in the bottom 11% of the window where the buttons are located.*

If the click was in the bottom 11% of the window, execution moves into the braces and the first inner **if** *(#2) is executed. This* **if** *uses the* **&&** *operator to see if the mouse was horizontally within the bounds of the red button. Both sub-expressions must be true for the test to be* **true**. *If the test is* **true**, *the tint is set to red. If the test of the first inner* **if** *(#2) is* **false**, *then execution moves to the next* **else-if** *(#3). The* **if** *checks to see if the click is horizontally within the green button. If the test is* **true**, *the tint is set to green. If it is* **false**, *execution moves to the next* **if-else** *(#4). The* **if** *tests to see if the user clicked horizontally in the blue button. If the test is* **true**, *the tint is set to blue.*

Notice that there is no fourth inner **if** *- just a stand-alone* **else** *(#5) which is only executed if all three of the inner* **if***s are* **false**. *Leaving out the fourth* **if** *is possible because the buttons occupy the entire width of the window. Had Jim drawn the buttons smaller than the width, the user could possibly click in the bottom 11% of the window but not on a button. In this case a fourth inner* **if** *to check for the NONE button would be required.*

Locating Places and Using Places on an Image

This refers to the class code Set 10 / Demo0:

Demo1 is showing you how to locate positions on an image. This code is different from what was done in class. First, this image has a clear copyright.

Let's walk through the code.

```
PImage p;
int mousePressedCount;

void setup( )
{
   size( 800, 600 );
   p = loadImage( "p.jpg");
   println( "Image width is " + p.width +
            " and height is " + p.height );
   image( p, 0, 0, width, height );
   fill( 255 );
   strokeWeight( 5 );
   fill( 255, 0, 0 );

   mousePressedCount = 0;
}
```

One difference is that there is a global variable of type **int** named **mousePressedCount** that is initialized to zero in the **setup()** function.

This variable is used here to label the **println** outputs.

The first thing to look at is the **println**. The **+** operator is used here but not for adding. When the **+** operator has numbers as operands, it adds. If one (or both) of it operands is inside " " marks, (this is called a **String**) the **+** operator does not add - it concatenates[1]. Processing literally looks up the values of the

[1] From Wikipedia: In computer programming, string concatenation is the operation of joining two character strings end-to-end. For example, the strings "snow" and "ball" may be concatenated to give "snowball".

variables and pastes them to the words in the quotation marks to create a new string.

The second thing to look at are the dots or periods (shown in purple):

```
println( "Image width is " + p.width + " and height is " + p.height );
```

This is Processing's form of possession. The variable **p** is really an object of the class **PImage** (more on this in several weeks). Unlike primitive variables like
int mousePressedCount which has three components:

- a name
- a type
- a value

Objects can have much more information associated with them. They can even have their own functions. Along with the actual picture that is displayed, the variable has its owns width and height values. To get to these in our code, we have to use a form of syntax that shows possession. In English we use the apostrophe (or single quote) to do this:

- Jim's class
- Da moose's musings
- The students' work

The apostrophe shows possession. The Processing equivalent is the dot or period. The code, **p.width**
gives us the value of the image's width. This is different than plain **width** which is our program's variable that stores the width of the window.

We can use this println to see the size of the image and to get an idea of the aspect ratio (width to height) that we need to keep in mind if we want to display the image in a reasonable manner.

The code in mousePressed is nothing special:

```
void mousePressed( )
{
   mousePressedCount++;
   println( "For mouse click # " + mousePressedCount +
            " -- mouseX: " + mouseX + " and mouseY: " +
             mouseY );
   point( mouseX, mouseY );
   text( mousePressedCount, mouseX, mouseY);
}
```

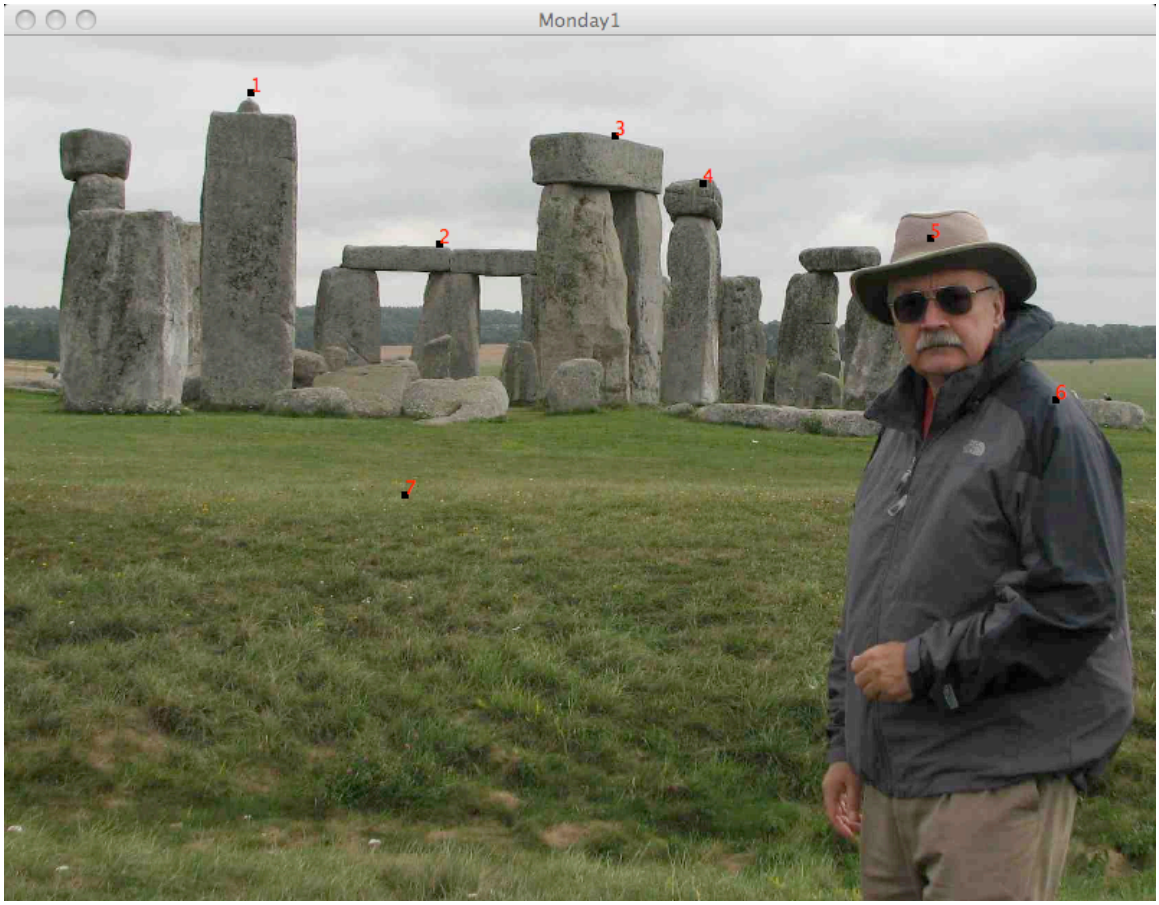
The first thing we see is that the value of the variable
mousePressedCount++;
is incremented by one. This is reasonable because the user has clicked the mouse.

Next there is a **println** *using the* **+** *operator to concatenate the output showing the click number and the location of the mouse.*

The code then draws a point on the screen where the mouse was clicked.

Finally, the point is labeled with the click number.

The next page shows the image and the console window during one run:




```
Image width is 1667 and height is 1218
For mouse click # 1 -- mouseX: 171 and mouseY: 39
For mouse click # 2 -- mouseX: 302 and mouseY: 144
For mouse click # 3 -- mouseX: 424 and mouseY: 69
For mouse click # 4 -- mouseX: 485 and mouseY: 102
For mouse click # 5 -- mouseX: 643 and mouseY: 140
For mouse click # 6 -- mouseX: 730 and mouseY: 252
For mouse click # 7 -- mouseX: 278 and mouseY: 318
```

The red numbers in the image correspond to the same mouse click numbers in the console window.

The program Demo2 just uses some recorded mouse click values (not the values shown above) to draw some vultures.

The program Demo3 shows a different method of animation that does not require the **frameCount** *or time. The animation is this code is controlled by the* **mousePressed( )** *function. It could also be controlled by a similar* **keyPressed()** *function.*

First, we see some global variables:

```
PImage p, p1;
int mousePressedCount;
```

The global variable **int mousePressedCount** *will be used to count mousePress events.*

The **setup()** *function is very similar to the one in Demo1 discussed above so we will skip over that function. Let's look at the* **mousePressed( )** *function:*

```
void mousePressed( )
{
  mousePressedCount++;
  if (mousePressedCount == 1)
  {
     image( p1, 301, 150-45, 40, 40 );
  }
   else if (mousePressedCount == 2)
  {
     image( p1, 238, 146-45, 40, 40 );
  }
   else if (mousePressedCount == 3)
  {
   image( p1, 169-10, 43-45, 40, 40 );
  }
   else if (mousePressedCount == 4)
  {
     image( p1, 59-20, 66-45, 40, 40 );
  }
   else if (mousePressedCount == 5)
  {
    image( p1, 404, 65-45, 40, 40 );
  }
  else
  {
     text( "   and it wasn't something good...", 20, 520 );
  }
}
```

There is no animation going on. The window is never "erased" with a new background or rectangle so we can use a very simple if/else structure.

This set of cascaded **if/else if/else** *… compares the value of the* **mousePressedCount** *variable to constants from 1 to 5. For the one value that is true, the* **p1** *image is displayed at the location of a former click.*

Using magic numbers like this is very bad. We will learn a better way to store these value very soon.

A New Form of Control - The Loop

This refers to the class code Set 10 Demo 4:

We have used one form of control - the **if/else**. *This is a form of control called selection or branching. This is discussed in Board Notes beginning on 0915 and continuing on 0917. You should refer to them if you are not comfortable with the* **if/else** *structure.*

A second group of control structures is the iterative group or the loops. The term loop is used because the syntax forces Processing to loop over a set of function calls until some condition is met. There are three forms of iteration available to Processing:

- *the while loop*
- *the for loop*
- *the do loop*

Processing ignores the do loop so we will do likewise.

The first loop is the while loop which has this form

```
While ( boolean expression )
{

}
```

A better explanation might look like this

```
While ( this evaluates to true )
{
   Do all of the stuff in here.
   When this is done,
        go back to the boolean expression and re-evaluate it
}
```

Here is the textbook naming of the parts of the while

```
While ( loop guard or loop test – a boolean expression )
{
    loop body – any valid Processing code.
}
```

As long as the loop guard or test is **true**, *the body of the loop is executed. Once the loop guard or test evaluates to* **false**, *the execution of the loop body stops.*

Here is a run of the code in Demo4:

The border of vultures is drawn by **while** *loops. Note that the entire border is drawn before Processing reveals the frame. Processing does not show us the frame until both loops stop and all of the code in* **draw()** *or in functions called by* **draw( )** *is finished.*

Here is the function that calls the function that draws the border of vultures:

```
void setup( )
{
   size( 800, 600 );
   p = loadImage( "p.jpg");
   p1 = loadImage( "vulture.jpg");

   println( "Width of vulture is " + p1.width +
            " and the height is " + p1.height);

   image( p, 0, 0, width, height );
   text( "Jim was beginning to get the idea that something
                  else was up...", 60, 500 );
   drawHengeVultures( );
   drawFrame( );  // draws the frame
}
```

and here is the code in the function **drawFrame( )**

```
void drawFrame( )
{
   // draw top and bottom frame row

  int x = 0;

 while( x < width )
 {
    image(p1, x, 0, width*.05, height*.06);
    image(p1, x, height -  height*.06,
                  width*.05, height*.06);
    x += int( width*.05 );
 }
```

```
 // draw side frames
 int y = 0;
   while ( y < height )
 {
    image(p1, 0, y, width*.05, height*.06);
    image(p1, width-width*.05, y,
              width*.05, height*.06);
    y += int( height*.06 );
 }
}
```

There are two loops. One draws the top and bottom rows of vultures and the other draws the vultures on both sides. (The other random vultures are drawn at the location of mouse clicks.)

Both use local variables initialized to zero:

```
  int x = 0;
```

This variable is local to this function and can only be used here. The value can be passed as a parameter. These two variables literally "walk" across and down the screen and are used as part of the anchor point for the image. Let's look at the first loop:

```
int x = 0;
while( x < width )
 {
    image(p1, x, 0, width*.05, height*.06);
    image(p1, x, height -  height*.06,
                 width*.05, height*.06);
    x += int( width*.05 );
 }
```

Actually, we begin before the loop with the declaration and initialization of the variable **x**.

Next we see the while loop syntax. The boolean expression that is the loop guard:

```
( x < width )
```

evaluates to true because the window is 800 pixels wide and the value of **x** *is zero. Since it is true execution moves into the loop body:*

```
{
   image(p1, x, 0, width*.05, height*.06);
   image(p1, x, height -  height*.06,
                width*.05, height*.06);
   x = x + int( width*.05 );
}
```

There are three statements in the body of the loop:

- *the first two draw the image of the vulture along the top and bottom of the window*
- *the third statement changes the value of* **x**

The third statement is vital. By changing the value of **x** *to a larger value, it will eventually get larger than the value of the width and the loop will stop or terminate.*

If we leave this line out, the loop will iterate infinitely which is usually a bad result.

This third line also "drives" the value of **x** *across the screen which means that we draw a new image at each new value of* **x**. *The result is a line of vultures at the top of the screen.*

Since both the top and bottom rows are the same length, we can draw both lines of vultures within this loop's body.

Look at the code in the second loop that draws the vertical lines of vultures. You will see a very similar structure but in this code, the variable is the **y** *value of the location of the image.*

We can place any valid Processing code within the body of the loop. Demo 5 is a slight modification of Friday4. Here is the output:

In keeping with the recent Valentine season, Jim has replaced the horizontal rows of vultures with rows of red and white hearts. Here is just the code that draws the hearts:

```
void drawFrame( )
{
  noStroke( );
  int counter = 0;
  int x = 0;

  while ( x < width )
  {
      if ( counter%2 == 0)
      {
         fill( 255, 0, 0 ); // red
      }
      else
      {
         fill( 255 );  // white
      }
      drawHeart(x, int(height*.03 ) );
      drawHeart(x, int(height-height*.03) );

      x += width*.06;
      counter++;
   }
```

Jim wanted to alternate between red and white hearts. There was no readily available data so he declared a variable named **counter** *to use just for the purpose.*

This determination is done in the **if/else** *that is inside the loop body:*

```
while ( x < width )
{
     if ( counter%2 == 0)
     {
        fill( 255, 0, 0 ); // red
     }
     else
     {
        fill( 255 );  // white
     }
```

This is a common way of alternating between two values. If the **counter** *is even (the* **if***'s test evaluates to true), the color is red. If the* **counter** *is odd, the color is white.*

One last item in case you have forgotten or missed class on the day this was disvussed:

This code above:

```
drawHeart(x, int(height*.03 ) );
drawHeart(x, int(height-height*.03) );
```

has code that results an integer value being copied into the corresponding argument in the function definition of **drawHeart(int, int).**

Processing as a function named **int( )** *which takes a* **float** *value as an parameter and it returns a truncated* **int** *value.*