Control

We need to give our programs the ability to make decisions. You may have experimented with this in homework #4 if your code decided what to do based on the user's keyboard input. But what if we need more… Suppose we have a menu of key inputs or variables have values that we do not want to use in our program. We need more tools in our tool kit. This capability requires the introduction of the control syntax.

Control refers to a group of syntactic structures in the language that allows the program to "control" or decide

- which functions to call and which ones to skip or if the value of a variable is acceptable.
- how many time to repeat a function call or group of function calls

The first bullet uses control structures we describe as "selection" or "branching" control structures. This is where we start. We will return to the second bullet's structures (the looping or iterative control structures) in a few days. The control syntax for selection is the if. It has several forms:

if (test or guard) **{** **do this if the test is true** **}** ***do nothing if the test is false*** ***Tthere is only one branch here – the if branch. Based on the test, Procssing will either select the if branch or select nothing***	**if (test or guard)** **{** **do this if the test is true** **}** **else** **{** **do this if the test is false** **}** ***There are two branchs here: the if and the else. Based on the test, Processing will select only one of these two branches. We say that, "The execution of the program flows through either the if or the else branch but not both of them."***

The test or guard must evaluate to a value of true or false. It cannot evaluate to an int or float value. It must be evaluate to true or false.

Expressions that evaluate to true or false are called boolean expressions in Processing. The b is lower case.

- We can declare boolean variables:
  boolean b = false;
- We can define functions that have a boolean return type (more on this next week…) :

```
boolean weHitTheTarget( int x, int y )
{
    . . .
}
```

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

- We can use boolean expressions with the relational operators that you used in your earlier math classes;

Operator	Meaning
==	Equality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal

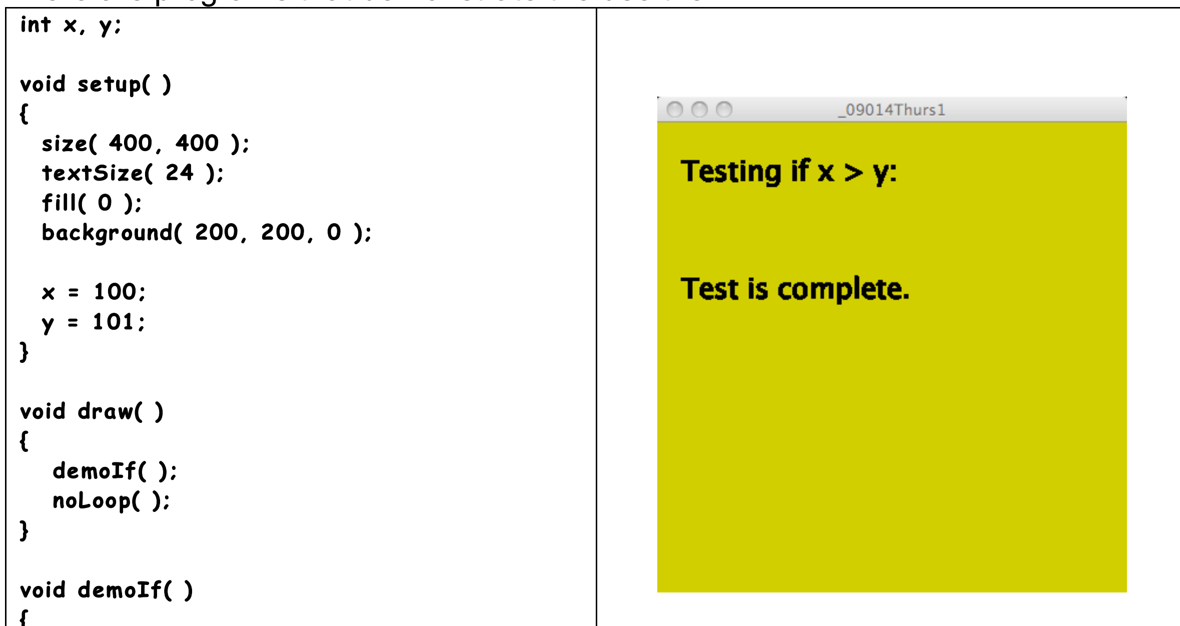
Note that the operators composed of two characters MUST NOT have a space between the two characters.
The equality operator is two equal signs. One equal sign is the assignment operator.

- We can also use the relational operators to combine Boolean expressions (see page 9).

Operator	Meaning
&&	AND
||	OR
!	NOT

For the next discussion of the if control structure below, we will use boolean expressions. In a later set of notes, we will use boolean variables and boolean functions.

Here are programs that demonstrate the use the if:



```
int x, y;

void setup( )
{
  size( 400, 400 );
  textSize( 24 );
  fill( 0 );
  background( 200, 200, 0 );

  x = 100;
  y = 101;
}

void draw( )
{
   demoIf( );
   noLoop( );
}

void demoIf( )
{
```

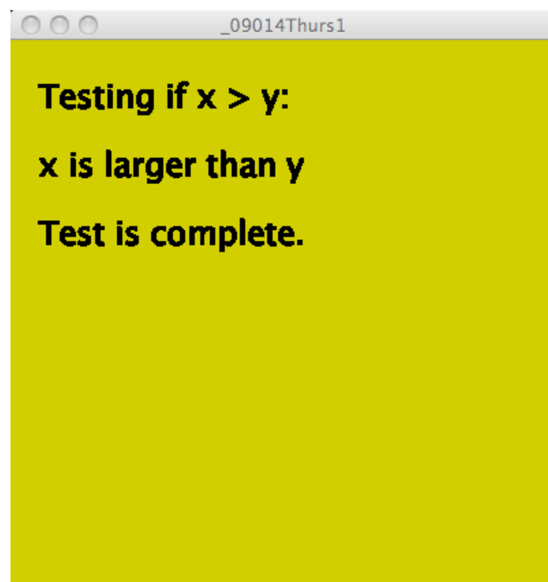

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

```
  text( "Testing if x > y: ", 20, 50 );
   if ( x > y )
   {
      text( "x is larger than y" , 20, 100 );
   }
   text( "Test is complete ", 20, 150 );
}
```

In the program above, the boolean expression that makes up test or guard of the if evaluates to false because x is less than y. Since the test is false, the code inside the braces is skipped. It is not selected. It is not executed.

In this next program, the value of x is larger than the value of y:

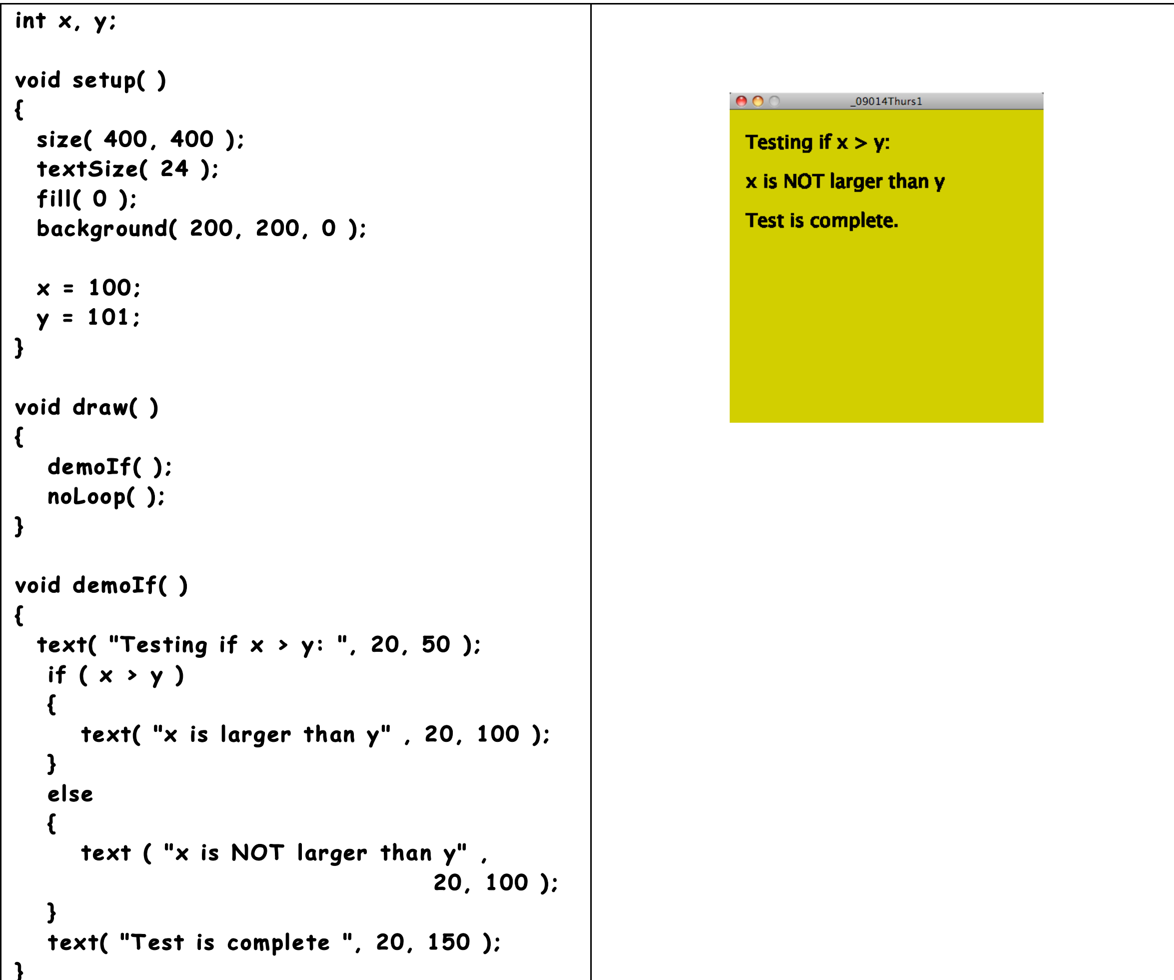
```
int x, y;

void setup( )
{
  size( 400, 400 );
  textSize( 24 );
  fill( 0 );
  background( 200, 200, 0 );

  x = 100;
  y = 101;
}

void draw( )
{
   demoIf( );
   noLoop( );
}

void demoIf( )
{
  text( "Testing if x > y: ", 20, 50 );
   if ( x > y )
   {
      text( "x is larger than y" , 20, 100 );
   }
   text( "Test is complete ", 20, 150 );
}
```



In the program above, the test evaluates to true so the code within the braces is selected. The code with the braces is executed and we see a different output.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

Next we switch to the other form: **if/else:**



```
int x, y;

void setup( )
{
  size( 400, 400 );
  textSize( 24 );
  fill( 0 );
  background( 200, 200, 0 );

  x = 100;
  y = 101;
}

void draw( )
{
   demoIf( );
   noLoop( );
}

void demoIf( )
{
  text( "Testing if x > y: ", 20, 50 );
   if ( x > y )
   {
      text( "x is larger than y" , 20, 100 );
   }
   else
   {
      text ( "x is NOT larger than y" ,
                                   20, 100 );
   }
   text( "Test is complete ", 20, 150 );
}
```


The value of x is less than y so the test is false. The program now has an else as part of the if. The else is executed when the test evaluates to false as shown in the output to the right.

Question: What happens if x and y are equal???
Code it and see before reading further. . .

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

Here is the code and how it is executed when the values of x and y are different

Code	Assume this: x = 200; y = 100	Assume this: x = 100; y = 200	Assume this: x = 100; y = 100
`int x, y;`			
`void setup( )`			
`{`			
`  size( 400, 400 );`			
`  textSize( 24 );`			
`  fill( 0 );`			
`  background( 200, 200, 0 );`			
`  x = 100;`			
`  y = 101;`			
`}`			
`void draw( )`			
`{`			
`   demoIf( );`			
`   noLoop( );`			
`}`			
`void demoIf( )`			
`{`			
`   text( "Testing if x > y: ", 20, 50 );`			
`   if ( x > y )`	true	false	false
`   {`			
`      text( "x is larger than y" , 20, 100 );`	do this	skip this	skip this
`   }`			
`   else if ( x < y )`	skip this	true	false
`   {`			
`      text ( "x is NOT larger than y" , 20, 100 );`		do this	skip this
`   }`			
`   else`		skip this	
`   {`			
`      text ( "x EQUAL to y" , 20, 100 );`			do this
`   }`	to here	to here	
`   text( "Test is complete ", 20, 150 );`	do this	do this	do this
`}`			

The third if is not needed. In this example there are only three possibilities: x > y, x < y, x == y. If the first two possibilities are false, then the third one must be true. This means that we do not need to make the third test:

```
if ( x == y )
```

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

Two more things:
We can put ifs within other ifs and other elses.

```
if (   )
{
   if(  )
   {

   }
   else
   {

   }
}
else
{
   if(  )
   {

   }
}
```

This is called nesting.
The if/else that is colored blue and red is the outer if (or outer if/else…)
The orange if/else is nested within the outer if.
The green if is nested within the outer else.

You can combine these in any reasonable way that is needed to solve the problem.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

We can also “cascade” a series of if/elses . You saw this previoulsy and it will be very common in our code.

```
if (   )
{

}
else if(  )
{

}
else if
{

}
else
{

}
```

When the test in the first if is true, the code inside the braces of the first if is executed and rest of the code in the entire if is finished. No more code (the tests or the code in the braces) is executed.

When the test in the first if is false, execution shifts to the test in the second if.

When the test of the second if is true, the code within the braces of the second if is executed and the rest of the code is skipped

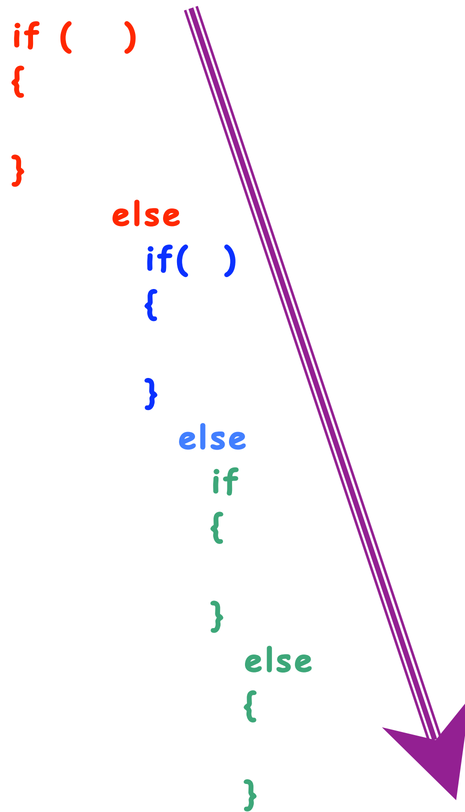
When the tests in both the first and second ifs are false, execution shifts to the third if.

When the test of the third if is true, the code within the braces of the third if is executed. When the test is false, the code within the else of the third if is executed.

You will see a cascaded if/else structure very soon in your code.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

You may be wondering why these are called cascaded. The reason is the way "we used to format them" in the code. In the distant past the code might have looked like this:

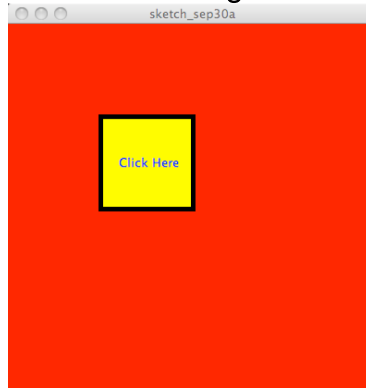


Sorta' like a waterfall. . .

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

The && Operator (The AND operator):

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.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

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

&&

This is two ampersands 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 seen by Processing as 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.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved

There is an OR operator.

Syntactically the OR operator 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.

Copyright Jim Roberts January 2013 Pittsburgh, PA 15221 All Rights Reserved