User Input

You have seen the **mousePressed()** and **keyPressed()** functions in class. These are functions that WE define but Processing calls when a key-press or mouse-press event occurs.

This "usually" works well but not always. As we should know by now, the **draw( )** function is repeated and each iteration produces a frame that we see in the graphics window. What happens if we hold down the mouse button or a key for several iterations? We need to do some coding to find out. Here is the code:

```
void setup( )
{
   size( 200, 400 );
}

void draw( )
{
    println( "in draw( ) : frameCount is " + frameCount);
}

void keyPressed( )
{
   println( "  in keyPressed( )  : frameCount is " + frameCount );
}
```

and here is part of the output when Jim held a key down for several seconds:

```
in draw( ) : frameCount is 74
in draw( ) : frameCount is 75
in draw( ) : frameCount is 76
in draw( ) : frameCount is 77
  in keyPressed( )  : frameCount is 77
in draw( ) : frameCount is 78
in draw( ) : frameCount is 79
in draw( ) : frameCount is 80
in draw( ) : frameCount is 81
in draw( ) : frameCount is 82
in draw( ) : frameCount is 83
in draw( ) : frameCount is 84
in draw( ) : frameCount is 85
```

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

Notice that the code in the **keyPressed( )** *function was executed but only for one iteration. Jim held the key down for at least two seconds or about 120 iterations. We might have expected to see this:*

```
 in draw( ) : frameCount is 77
  in keyPressed( )  : frameCount is 77
in draw( ) : frameCount is 78
  in keyPressed( )  : frameCount is 78
in draw( ) : frameCount is 79
  in keyPressed( )  : frameCount is 79
...
in draw( ) : frameCount is 197
  in keyPressed( )  : frameCount is 197
in draw( ) : frameCount is 198
  in keyPressed( )  : frameCount is 198
```

So we can conclude that the **keyPressed( )** *function is called one time for each press during the frame in which it is pressed. The event is ignored in subsequent frames so that holding down the key does not cause the function to be repeatedly called and executed.*

The same is true for the **mousePressed( )** *function.*

But what if we want to do something as long as a key is held down or the mouse button is pressed. Asking the user to press and release the key or button 60 times per second sounds a bit unreasonable. . .

Oh what shall we do... ???

The answer is held by two system variables that are of the type, **boolean**. *These are named:*

keyPressed *and* **mousePressed**

The are not functions - there are no parentheses . . .

Since these variables are **boolean** *variables, the have either the value of* **true** *or* **false**:

- **true** *when the button or a key is down*
- **false** *when the button or all keys are up*

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

So, we can use them to see if either the button or a key is down.

Here is the Demo 5 code programs in Class Code Set 09

```
// Set 09 Class Code
// Demo 5

void setup( )
{
   size( 400, 400 );
   textSize( 14 );
   textAlign( CENTER, CENTER );
   stroke( 0, 0, 255 );
   strokeWeight( 5 );
   background( 255 );
}

void draw( )
{
   fill( 0 );
   text( "Press a Key  ...", width/2, 30 );
   stroke( 0, 0, 255 );

   checkKey( );
   checkMouse( );
}

void checkMouse( )
{
   if ( mousePressed == true )
   {
     line( mouseX, mouseY, pmouseX, pmouseY );
   }
}

void checkKey( )
{
  if ( keyPressed == true )
  {
     stroke( 255, 0, 0 );
  }
}
```

This first part is the usual stuff

For each iteration of draw we tell Processing to execute these two functions

This is the definition of checkMouse( ) where we test the system variable mousePressed to see if it is true. If it is true, we draw a line from the location of the mouse in this frame back to the location of the mouse in the previous frame.

If the mouse button is not down, we draw nothing. This works for every frame the button is down.

We do the same thing in this function using the keyPressed variable. If any key is down, we change the color of the line to red.

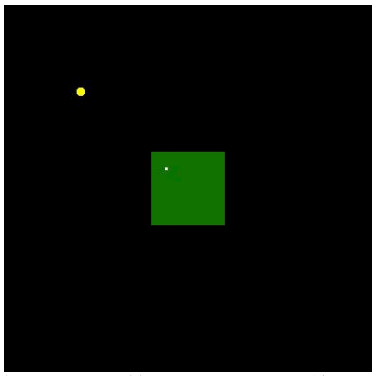
You should use the **keyPressed( )** *and* **mousePressed( )** *function if they will work and use the* **mousePressed and**

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

keyPressed( ) variables only when the function does not accomplish what you need to do.

Here is the Demo 4 code programs in Class Code Set 09
Demo 4 introduced you to the **map( )** function. The **map()** function is very helpful in many ways. We can use it to scale a value from one range to another. Probably our most common mapping is temperature when we convert temperatures in degrees Fahrenheit to degrees in Celsius. We also map when we convert currency from one form to another.

Mapping in our programming often uses the mouse location in the window to determine a value for color or rotation. We will do color next. Demo 5 mapped the mouse's location in the window to its corresponding location in a box inside the window.



The larger yellow dot is the actual mouse location and the smaller white dot is the mapped mouse location. The mapped location is proportionally the same in the green rectangle as the actual location is in the entire window.

The **map( )** function requires five arguments of either **float** or **int**. Here is the line of code that maps the **x** coordinate of the white dot based on the **x** coordinate of the yellow dot:

```
    argument #      1     2   3        4                  5
  int x = int( map( mouseX, 0, width, smallRectX, smallRectX+smallRectDim) );
```

Let's take the arguments one at a time:

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

Arg 1 *is the value we need to map - in this case it is the* **x** *coordinate of the mouse.*
Arg 2 *is the smallest value the mouse can have -- in this case this is the left edge of the window with a coordinate value of* **zero.**

Arg 3 *is the largest value the mouse can have -- which is the right edge of the window. It has a coordinate value of* **399.**

Arg 4 *is the smallest value of the range into which we are mapping* **arg1.** *The inner box's left edge has a coordinate value of* **smallRectX** *pixels, which is 160.*

Arg 5 *is the smallest value of the range into which we are mapping* **arg1.** *The inner box's right edge has a coordinate value of* **(smallRectX + mallRectDim)** *pixels, which is 280.*

```
// Friday February 11  Class Code
// Demo 5
// map function

color bigRectColor, smallRectColor;
color actualMouseLocationColor,
mappedMouseLocationColor;
int smallRectX, smallRectY, smallRectDim;

void setup( )
{
  size( 400, 400 );

  bigRectColor = color( 0 );
  smallRectColor = color(18, 98, 3 );

  actualMouseLocationColor =
         color( 255, 255, 0 );
  mappedMouseLocationColor =
         color( 255 );

  smallRectX = int(width*.4);
  smallRectY = int(height*.4);
  smallRectDim = int(width*.2);


  noCursor( );
}
```

This is the usual stuff for a while

Until we get to here.

There are several different forms of cursor we can use and we can turn it off completely. Check the API for cursor.

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

```
void draw( )
{
   background( bigRectColor);
   fill( smallRectColor);
   noStroke( );
   rect( smallRectX, smallRectY,
          smallRectDim, smallRectDim);

   if ( mousePressed == true)
   {
     showActualMouseLocation( );
     showMappedMouseLocation( );
   }
}

void showActualMouseLocation( )
{
   stroke( actualMouseLocationColor );
   strokeWeight( 8 );
   point( mouseX, mouseY );
}

void showMappedMouseLocation( )
{
   stroke( mappedMouseLocationColor );
   strokeWeight( 3 );
   int x =
     int(

       map( mouseX,
       0, width,
       smallRectX, smallRectX+smallRectDim) );


   if ( x < smallRectX )
   {
      x = smallRectX;
   }
   else if ( x > smallRectX+smallRectDim )
   {
      x = smallRectX+smallRectDim;
   }

   int y = int(
    map( mouseY,
         0, height,
         smallRectY smallRectY+smallRectDim) );
   if ( y < smallRectY )
   {
      y = smallRectY;
   }
   else if ( y > smallRectY+smallRectDim )
   {
      y = smallRectY+smallRectDim;
   }
   point( x, y);
}
```

We only map the mouse when the mouse button is pressed.

This is called when the mouse button is pressed.
It sets the color and size of the stroke and draws a point at the location of the mouse in the window.

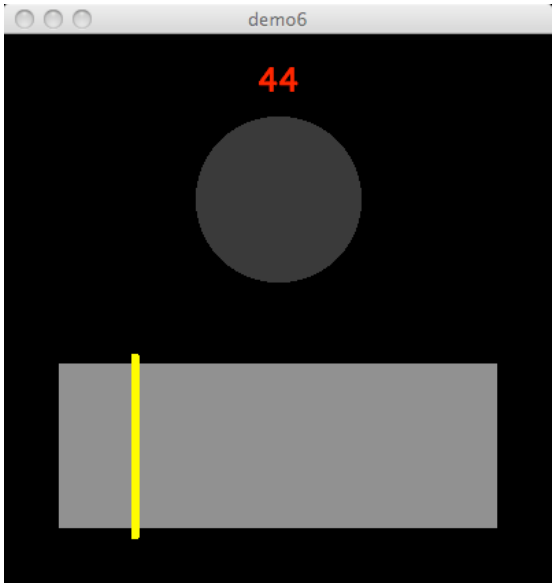
This is also called when the button is pushed.
It does the mapping of the mouse's location into the small green rectangle.
The map( ) function returns a float so we have to use the int function to convert the value to an int.
We are mapping mouseX
which has a value between zero and 400
inside the green rect which is 160 to 280.

The map( ) function does not limit the values - it maps them. Jim's testing found that the mouse was tracked outside the window on his machine so he added this if/else if to keep the mapped value between 160 and 280.

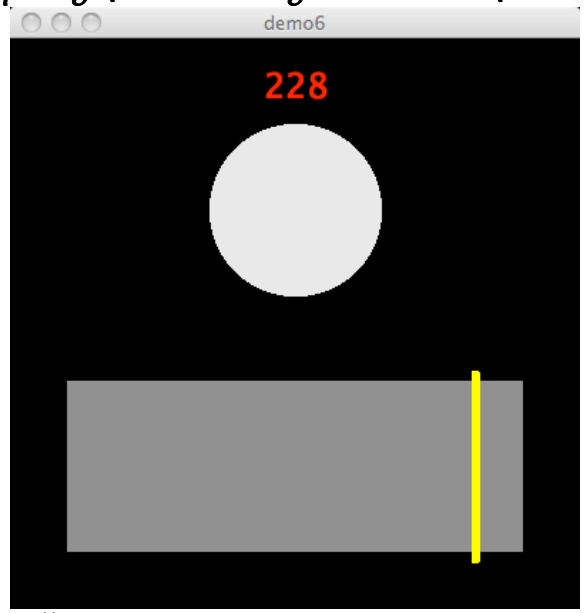
This is the same code mapping the mouse's y location into the green rect.

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

Here is the Demo 6 code programs in Class Code Set 09
Demo 6 mapped the mouse location into a color value for gray AND mapped the color into the location fo the yellow line. A Quick Refresher - zero is black, 255 is white, all values in between are gray. Here is the mapping for a small value of **mouseX**:



Here is a mapping for a large value of **mouseX**:



The code is on the next page:

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

```
// Friday February 11  Class Code
// Demo 6
// map function

color circleColor;
int circleX, circleY, circleDiameter;
int colorValue;
int barX, barY, barWidth, barHeight;

void setup( )
{
   size( 400, 400 );
   circleX = width/2;
   circleY = int (height*.3);
   circleDiameter = int(width *.3);
   colorValue = 127;
   circleColor = color( colorValue );
   barX = int( width*.1);
   barY = int( height*.6);
   barWidth = int( .8*width);
   barHeight = int( .3*height );
}

void draw( )
{
   background( 0);
   circle( );

   if (mousePressed)
   {
      mapMouse( );
   }
   drawColorBar( );
   showData( );
}

void mapMouse( )
{
   colorValue = int( map( mouseX,
                barX, barX + barWidth,
                0, 255 ) );


   if ( colorValue < 0 )
   {
      colorValue = 0;
   }
   else if ( colorValue > 255 )
   {
      colorValue = 255 ;
   }
   circleColor = color( colorValue );
}



void drawColorBar( )
{
```

Usual stuff

Lotsa' global variables to keep the code readable. . .

We only alter the color value if the mouse button is down.

This maps the mouseX value into a color value between 0 and 255.

If the user moves outside the bar, the mapping can do strange things so we limit the upper and lower values of the color to 0 and 255.

Finally we set the value of the color variable.

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved

```
    noStroke( );
    fill( 127 );
    rect( barX, barY, barWidth, barHeight);
    stroke( 255, 255, 0 );
    strokeWeight( 5 );

    float x =
        map( colorValue,
             0, 255,
             barX, barX+barWidth) ;


    line( x, barY - 5, x, barY + barHeight + 5 );
}

void circle( )
{
   noStroke( );
   fill( circleColor);
   ellipse( circleX, circleY,
            circleDiameter, circleDiameter );
}
```

This is drawing the gray bar and a yellow mark to show where the color value is in the overall range from 0 to 255.

The map function is converting the current colorValue which is between 0 and 255 into an X value in the range between the left and right edges of the bar.

Once the mapping is done, we can draw the yellow line.

You may need to use the map function in some manner of your choosing to allow the user to control some aspects of the animation in your future work.

Get the class code and play with it to do different things. Get to one of us soon if you are not sure what is happening in this code.

Copyright Jim Roberts May 2011 Pittsburgh, PA 15221 All Rights Reserved