cis3.5, spring 2009, lab II.1 / prof sklar.

1 Getting started with Processing

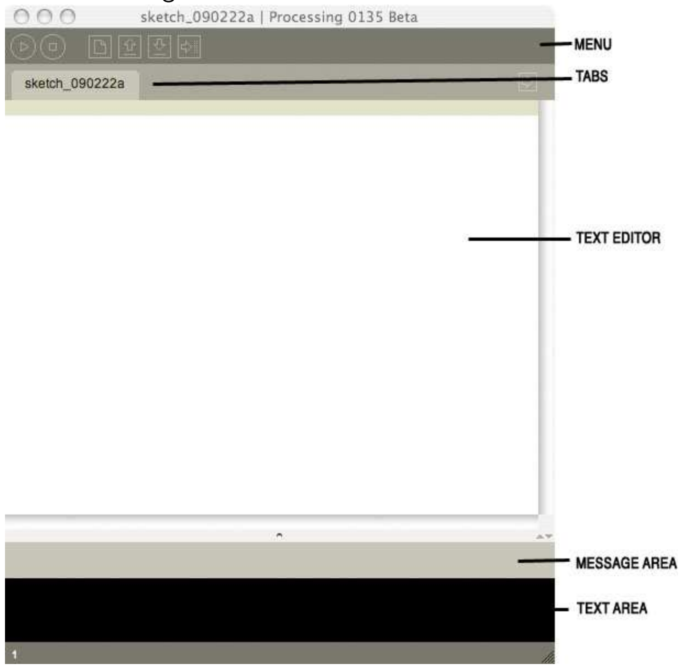
Processing is a "sketch" programming tool designed for use by non-technical people (e.g., artists, designers, musicians). For technical people, it is a handy tool for prototyping applications in Java.

You can do lots of things with Processing. This lab will focus on graphics and games.

1.1 Start up Processing

Double-click the **Processing** icon, which probably looks something like this:

The Processing window looks like this:



Note the annotations on the right in the image above that point out the areas of the window.

Menu buttons:

▷	**run**	compiles the code, opens a display window and runs the program.
□	**stop**	terminates a running program.
	new	creates a new "sketch" in the current window.
⇧	**open**	provides a menu with options to open files from your "sketchbook", an example or another a sketch on your computer.
⇩	**save**	saves the current sketch with its current name and location.
⇨	**export**	exports the current sketch as a **Java** "applet".

Notes on Menu buttons:

- **run** — Hold down the **shift** key to **Present** instead of run
- **new** — To create a new sketch in its own (new) window, use **File** - **New**
- **open** — Note that opening a sketch from the toolbar will replace the sketch in the current window. To open a sketch in a new window, use **File** - **Open**.
- **save**— If you want to give the sketch a different name, use **File** - **Save As**.

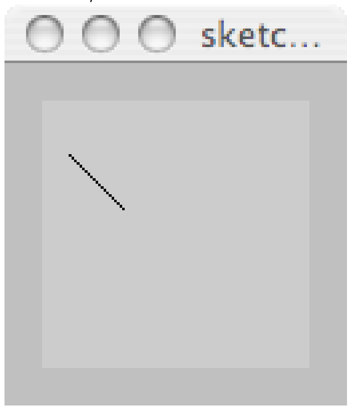
1.2 Write your first program: drawing a line

In the **text window**, type the following:

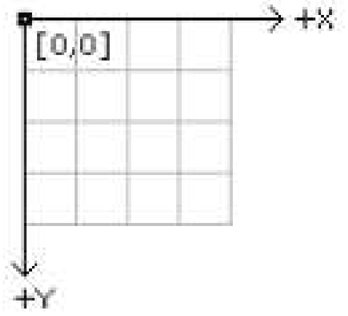
```
line( 10, 20, 30, 40 );
```

- Note the punctuation—parenthesis, commas and semi-colon.
- Note that the "function" **line** is written in *lower case*. Processing is case-sensitive, so watch the case with each new function introduced.

After you type the above in the text window, click on the **run** button. Processing will open a display window, like this:



What have you done?
The **line()** function takes four "arguments". These are the endpoints of a line. Imagine that the display window is a piece of graph paper with two axes: x and y. The x axis runs horizontally along the display window, starting with 0 on the left and increasing as you move to the right. The y axis runs vertically down the display window, starting with 0 on the top and increasing as you move down.



So, you have just drawn a line from $(x_1, y_1) = (10, 20)$ to $(x_2, y_2) = (30, 40)$.

1.3 Modify your program

- Try changing the values of the arguments to the **line()** function to different (x, y) values. Each time you change the values, click on the **run** button to see the effect of what you've done.

- Try adding a second **line()** function (put it on another line in the text editor, below the first **line()** function).

1.4 Adding color

- The Processing function for drawing in color is called **stroke()**. It takes one argument, a 6-digit *hexademical* number specifying the amount of *red*, *green* and *blue* in the color that should be used for drawing. For example:

  ```
  stroke( #ff0000 );
  line( 10,10,20,20 );
  ```

  Does this remind you of setting colors in HTML/CSS?
- Try adding a **stroke()** function call above your call(s) to **line()** in the text editor.
- Check out the command **Tools - Color Selector**. You'll find help for picking cool colors!
- Now try adding multiple **stroke()** calls to your sketch, one before each **line()** call, each one with a different color. This way, each line you draw will be a different color.
- Try drawing a green square using one call to **stroke()** and four calls to **line()**.
  Hint: plan out your code ahead of time by drawing a square on paper and figuring out what the coordinates of each of the four corners should be.
- Try drawing a square with each side a different color. You will need four calls to **stroke()** and four calls to **line()**.

1.5 Learning new functions

Look up the syntax of each of the following functions on the Processing on-line reference page:
`http://www.processing.org/reference/index.html`
Try putting each command in your sketch and see how they work.

- **rect()**
- **ellipse()**
- **fill()**
- **triangle()**
- **strokeWidth()**

1.6 Exporting an applet

Try clicking on the **export** button or selecting **File - Export**.
This will create a Java "applet" out of your sketch. A Java applet is embedded in an HTML file. With the export command, Processing saves the Java applet as well as an HTML called **index.html**. Both are placed in a subfolder called **applet** in the sketch's folder inside your sketchbook. Processing will open this subfolder. Click on **index.html** to view the Java applet version of your sketch.

2 Making your program interactive

In the first section (above), you learned how to write programs that only had **output**. This means the programs display stuff. In this section, you will learn how to write programs that take **input**, which means that you can have the program respond to things the user/viewer does.

As above, remember to be mindful of upper and lower case, as well as punctuation.

2.1 Create a new sketch and enter the following code in your text window.

```
void setup() {
  background( #ff0000 );
}

void keyPressed() {
  background( #0000ff );
}

void draw() {
}
```

Click on the **run** button, and then when Processing opens the display window, click anywhere in the display window and then click on any key. The color of the display window should change from red to blue.

The code in the **setup()** function runs as soon as the sketch starts.

The code in the **keyPressed()** function runs as soon as the user presses a key. Note that you (the user) have to click in the display window to give it *focus*, so that the sketch will recognize (i.e., be "listening") when a key is pressed.

2.2 The above program demonstrates input from the keyboard, when *any* key is pressed. Now try entering and running the code below, which responds differently when different keys are pressed.
Note: The program will respond to R, G, B, and W.
Don't forget to click in the display window, to give it focus, before pressing any keys.

```
void setup() {
  background( #000000 );
}

void keyPressed() {
  if ( keyCode == 'R' ) {
    background( #ff0000 );
  }
  else if ( keyCode =='G' ) {
    background( #00ff00 );
  }
  else if ( keyCode == 'B' ) {
    background( #0000ff );
  }
  else if ( keyCode == 'W' ) {
    background( #ffffff );
  }
}

void draw() {
}
```

After you have run the program, go back and look at the code. You will see the words **if** and **else**. These are called *control structures*, and they control the flow of the code. If the user presses the R key, one thing happens (what is it?); otherwise, if the user presses the G key, something else happens (what is it?); and so on. The **if...else** is called a *conditional* control structure, because it specifies what the program should do under conditions specified by the programmer.

- Try changing the code by modifying what happens when the user presses R. Note that whatever code you add/change, all functions have to be contained within curly brackets ({ and }). Currently, only the statement `background( #ff0000 );` is in between the curly brackets. If you add more lines of code, keep them between the same curly brackets.
- Now try changing the code by adding another condition of your own, when another key is pressed (other than R, G, B, or W).

3 Adding animation

Up until now, anything that we have had the program output has been *static*, in that it does not change by itself. In the second section, above, we gave the user some control to make changes in the display. In this section, you'll learn how to make things change in the display by themselves. Essentially, this is *animation*.

The basic principle behind animation is like that of an old-fashioned *flip book*. If you don't know what a flip book is, you can see a sample here: `http://www.flippies.com/flipbooks-gallery/`

In Processing, the idea is that your program will draw an object in the display window, wait a fraction of a second or so, and then clear the display and draw the object again, in a slightly different place. This will make it look like the object is moving across the display!

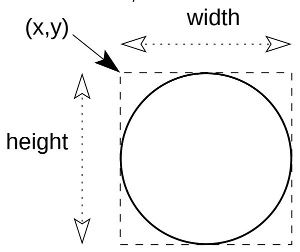
3.1 Enter this sample code into your text window.

```
int x;
int y;

void setup() {
  background( #000000 );
  ellipseMode( CORNER );
  x = 0;
  y = 50;
}

void draw() {
  background( #000000 );
  ellipse( x, y, 40, 40 );
  x = x + 1;
  if ( x > width ) {
    x = 0;
  }
}
```

Run the code and watch what happens.
Then turn the page...

The **draw()** function is causing a shape (an ellipse) to be drawn in the display window. The ellipse moves horizontally across the window.

The figure below illustrates how an ellipse is defined in Processing. The ellipse is shown with a dark line, and its *bounding box* is shown in a dashed line. The ellipse is defined by specifying the (x, y) coordinates of the upper left corner of the bounding box, plus the *width* and *height* of the bounding box. Note that, as in this case, when the width and height are equal, the ellipse is actually a circle.



There are many things to notice about this program:

- The animation happens in the example because each time the ellipse is drawn, the x value of the upper left corner of the bounding box has changed.
- The **draw()** function is called automatically by Processing every 1/60th of a second. This means that if you change where the ellipse is drawn each time that **draw()** is called, it will look like animation :-)
- The number of times per second that **draw()** is invoked is called the *frame rate*. The default is 60 frames per second, or 1/60 of a second. If you want your animation to move faster, you can increase the frame rate. If you want your animation to move slower, you can decrease the frame rate. The **frameRate()** function can be invoked to set the frame rate. Place a call to this function within the **setup()** function.
- The first line of the program looks like this:

  ```
  int x;
  ```

  This is called *declaring a variable*. A *variable* is a special component of a Processing program, and indeed many types of programs written in other languages. Variables give a programmer a way to save a value and use it somehow to control what the program does. In this case, we are using it to control where the ellipse is drawn.

  Variables are given a "data type", in this case **int**, which stands for integer (i.e., a whole number). Variables are also given a "name", in this case **x**. Finally, variables are given a "value". In this case, x is initially given the value zero, like this: $x = 0$ inside the **setup()** function.

  Inside the **draw()** function, the value of x changes ($x = x + 1$), each time the function is called. Note that there is an **if** statement which checks to see if the value of x is bigger than the *width* of the display window. If it is, then the program re-sets the value of x to 0, which means that the ellipse will "wrap around" the right side of the window and appear on the left side. Run the program for a few seconds to watch that happen.

Try modifying the code to make the ellipse move vertically instead of horizontally.
Hint: modify y instead of x.

Try modifying the code again to make the ellipse start with a small width and height (e.g., instead of $40, 40$, start with $1, 1$) and grow larger each time **draw()** is called.
Decide what you want to do when the ellipse has grown to be too big to fit inside the display window. Note that the Processing system variable **width** is defined to be the width of the display window and the Processing system variable **height** is defined to be the height of the display window.

On-line references

Processing

- `http://www.processing.org/` (main processing web site)
- `http://www.processing.org/reference/index.html` (on-line reference)

Processing for mobile devices:

- `http://mobile.processing.org/`