1 Getting started with Processing

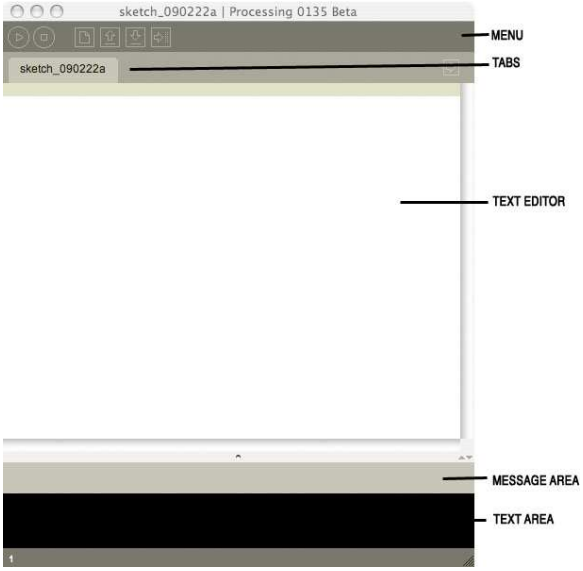
Processing is a "sketch" programming tool designed for use by non-technical people (e.g., artists, designers, musicians). For technical people, it is a handy tool for prototyping applications in Java.

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

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.

Description of menu buttons:

run Compiles the code, opens a display window and runs the program.
Hold down the **shift** key to **Present** instead of run

stop Terminates a running program.

new Creates a new "sketch" in the current window.
Hold down the **shift** key to **Present** instead of run

open Provides a menu with options to open files from your "sketchbook", an example or another a sketch on your computer.
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 Saves the current sketch with its current name and location.
If you want to give the sketch a different name, use **File** - **Save As**.

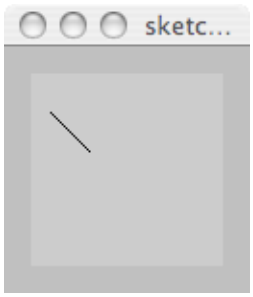
export Exports the current sketch as a **Java** applet.

1.2 *Write your first program in Processing—Draw a line*

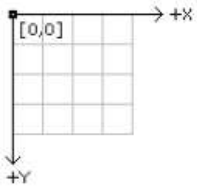
Once you have started up Processing, find the **text window**, which is really a text editor.
In the text window, type the following:

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

After you type the above in the text window, click on the **run** button.
Processing will open a display window and draw the line you specified, like the image on the left, below:



The Processing **line()** function takes four arguments: the endpoints of a line extending from (x_1, y_1) to (x_2, y_2). So, you have just drawn a line from $(x_1, y_1) = (10, 20)$ to $(x_2, y_2) = (30, 40)$. The Processing display window follows standard computer graphics conventions for a 2-dimensional coordinate system. 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.



Click on the **stop** button to stop your sketch running.

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. *(This is the same way colors are defined in HTML and CSS.)* For example:

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

- 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()**.

2 Quick overview of Processing syntax

Processing syntax is similar to Java and C/C++. *(We assume here that you are familiar with Java or C/C++.)*

Processing code is case-sensitive. Statements end with a semi-colon (`;`). Function arguments are surrounded by parentheses, `(` and `)`. Blocks of code, including function bodies, are surrounded by curly brackets, { and }.

Comments are enclosed in `/*...*/` or appear on a line after `//`.

Operators.

- arithmetic: $+$, $-$, $*$, $/$, $\%$
- comparison: $>$, $<$, $>=$, $<=$, $==$, $!=$
- logic: $||$ (logical OR), $\&\&$ (logical AND), ! (logical NOT)
- bitwise: $|$ (bitwise OR), $\&$ (bitwise AND), $<<$ (shift left), $>>$ (shift right)

Conditional control structures.

- `if...else`
- `switch...case...default...break...continue`

Repetition control structures.

- `for` loops
- `while` loops

Data types.

- primitive types: `boolean`, `byte`, `char`, `int`, `float`, `color`
- conversion functions: `boolean()`, `byte()`, `char()`, `int()`, `float()`, `str()`
- composite types: `Array`
- object types: `String`, `PImage`

Array handling.

- `new` (function, e.g., `int[] d = new int[10];`)
- `length` (variable, e.g., `d.length`)
- `[...]` (indexing, e.g., `d[3]`)

String objects.

- functions: `String.charAt()`, `String.equals()`, `String.indexOf()`, `String.length()`, `String.subString()`, `String.toLowerCase()`, `String.toUpperCase()`

PImage objects

- variables: `width`, `height`, `pixels[]`
- functions: `PImage.get()`, `PImage.set()`, `PImage.copy()`, `PImage.mask()`, `PImage.blend()`, `PImage.filter()`, `PImage.save()`, `PImage.resize()`, `PImage.loadPixels()`, `PImage.updatePixels()`

You can look up the syntax of each of these functions (and more!) on the Processing reference page:

`http://www.processing.org/reference/index.html`

This is also accessible using the **Help – Reference** options from the Processing menu.

Note that the functions listed throughout this lab are only a small sample of what exists in the Processing environment. You are encouraged to visit the reference pages and find more functions to help you do what you'd like your program to do. Before you write a function yourself from scratch, be sure to check out what is already part of the language!

3 Output

3.1 Drawing different kinds of shapes

In addition to lines, you can use Processing to draw all kinds of shapes. Modify your program to draw different shapes using the functions listed below:

- **point()** – draws a point
  syntax: `point( x, y );`
- **triangle()** – draws a triangle
  syntax: `triangle( x1, y1, x2, y2, x3, y3 );`
- **quad()** – draws a quadrilateral
  syntax: `quad( x1, y1, x2, y2, x3, y3, x4, y4 );`
- **rect()** – draws a rectangle
  syntax: `rect( x, y, width, height );`
- **ellipse()** – draws an ellipse
  syntax: `ellipse( x, y, width, height );`
- **arc()** – draws an arc
  syntax: `arc( x, y, width, height, start, stop );` *where* `start` *and* `stop` *are in radians*

Note that the (x, y) coordinates of each shape are either the center of the shape or the upper left corner of the shape's bounding box—depending on the **mode** for drawing each shape. The mode can be set using one of the following functions:

- **rectMode()** – sets mode for drawing rectangles
  syntax: `rectMode( MODE );` *where* `MODE`:
  - = `CORNER` (default): (x, y) are the upper left corner of the rectangle's bounding box and $(width, height)$ are the extent of the rectangle's bounding box
  - = `CENTER`: (x, y) are the center of the rectangle's bounding box and $(width, height)$ are the extent of the rectangle's bounding box
  - = `RADIUS`: (x, y) are the center of the rectangle's bounding box and $(width, height)$ are half the extent of the rectangle's bounding box
  - = `CORNERS`: (x, y) are the upper left corner of the rectangle's bounding box and $(width, height)$ are the lower right corner of the rectangle's bounding box
- **ellipseMode()** – sets mode for drawing ellipses and arcs
  syntax: `ellipseMode( MODE );` *where* `MODE`:
  - = `CENTER` (default): (x, y) are the center of the ellipse and $(width, height)$ are the diameters of the ellipse
  - = `RADIUS`: (x, y) are the center of the ellipse and $(width, height)$ are the radii of the ellipse
  - = `CORNER`: (x, y) are the upper left corner of the ellipse's bounding box and $(width, height)$ are the diameters of the ellipse
  - = `CORNERS`: (x, y) are the upper left corner of the ellipse's bounding box and $(width, height)$ are the lower right corner of the ellipse's bounding box

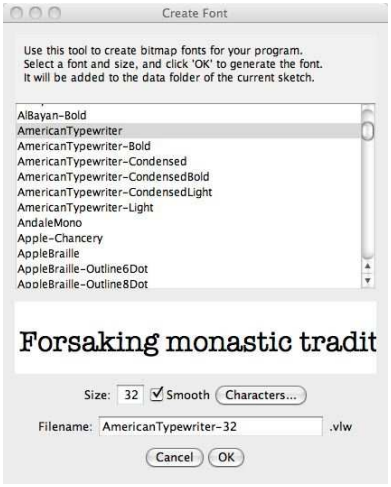
You can change the style with which shapes are drawn using the following functions:

- **fill()** – sets the color used to fill shapes
- **strokeWeight()** – sets the width of the stroke used for lines, shapes and points
- **strokeJoin()** – sets the style of joints where lines meet
- **strokeCap()** – sets the style of line endpoints

3.2 Drawing text

You can write strings on your Processing sketch by first creating a font, then loading the font into your sketch and drawing the text string in a location in your display window.

(a) First, you need to create the font.
Select **Tools – Create Font...** from the Processing menu. A window will pop up with a scrollable list of fonts, like the one shown below.



There is a preview of the font toward the bottom of the window.

Select the font you'd like to use (by clicking on the name of the font in the scrollable list).

Set the size of the font (by typing a number in the **Size:** field).

Note the value in the **Filename:** field at the bottom of the window. Then click on **OK**.

In the example, the font chosen is **AmericanTypewriter** and its size if **32**. The corresponding file name is: `AmericanTypewriter-32.vlw`

(b) Second, modify your code in the text editor to use the font you created.
Declare and instantiate a `PFont` object. This is where you load the font using the filename you just created, above:

```
PFont font = loadFont( "AmericanTypewriter-32.vlw" );
```

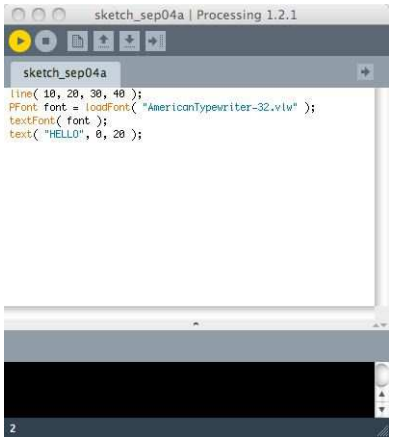
Select the font:

```
textFont( font );
```

Use the font:

```
text( "HELLO", 0, 20 );
```

A sample is shown below. The code is on the left, and the resulting sketch is on the right.



4 Program structure

So far, you have written simple Processing code that is comprised of a series of function calls. But Processing is a structured object-oriented language. You can declare variables, write your own functions and create your own objects.

4.1 The Draw() loop

There are two functions built into Processing that define the basic program structure: **setup()** and **draw()**. The **setup()** function is invoked once, when a sketch starts running. The **draw()** function is invoked multiple times—initially, right after the call to **setup()**, and then repeatedly until the sketch stops running.

If you know Java, then you may have figured out that a sketch is really a `Thread`*. The* **draw()** *function is like the* **Thread.run()** *method.*

Here is the code from the previous example re-written using the **setup()** and **draw()** functions:

```
void setup() {
  PFont font = loadFont( "AmericanTypewriter-32.vlw" );
  textFont( font );
}

void draw() {
  line( 10, 20, 30, 40 );
  text( "HELLO", 0, 20 );
}
```

The **draw()** function in the example above is constant—every time it is invoked, it does exactly the same thing. But if you want to create a dynamic sketch, one that includes animation and/or responds to user input, you will see where this program structure is useful. These functionalities will be discussed in the next sections, below.

4.2 Objects

You can create objects in Processing just as you can in Java and in C++. An example is shown below.

The class is defined at the bottom of the program, using the Processing keyword `class` followed by the user-defined name of the object being created (in this case, the name is "Spot"). An object of this type is declared at the top of the program, as a global variable. The object is instantiated in the **setup()** function. The object is drawn, using its own **Spot.display()** function, which is called in the program's **draw()** function.

```
Spot p; // declare the object
void setup() {
  p = new Spot( 35, 50 ); // instantiate the object
}
void draw() {
  p.display();
}
class Spot { // define the class
  int x, y;
  Spot( int x0, int y0 ) {
    x = x0;
    y = y0;
  }
  void display() {
    fill( 175 );
    ellipse( x, y, 10, 20 );
  }
}
```

5 Input

User input to Processing sketches can come from the **Keyboard** and the **Mouse**. We'll explore both here.

5.1 Keyboard Input

Below are the Processing variables and functions that handle keyboard input. The functions are listeners and should be overridden if used.

- **key** – character value of key used most recently; or = `CODED` if a special key was pressed (see below)
- **keyCode** – indicates which special key was pressed: `UP`, `DOWN`, `LEFT`, `RIGHT`, `ALT`, `CONTROL`, `SHIFT`
- **keyPressed** – boolean value set to `TRUE` if a key was pressed
- **keyPressed()** – function that is called when a key is pressed
- **keyReleased()** – function that is called when a key is released

Below is an example sketch that demonstrates keyboard input:

```
void setup() {
  background( #ff0000 );
}
void keyPressed() {
  background( #0000ff );
}
void draw() {
}
```

Type this code in your text window. Click on the **run** button, and then when Processing opens the display window, click anywhere in the display window–which places the "focus" in that window–and then click on any key. The color of the display window should change from red to blue.

The above program demonstrates input from the keyboard, when *any* key is pressed. Now try modifying the **keyPressed()** function, as below, to respond differently when different keys are pressed.

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

(5.1.a) Try modifying the code so that it responds to both uppercase and lowercase input (e.g., R and r).

(5.1.b) Try changing the code so that different shapes are drawn when the user presses each letter. For example, pressing R would cause the program to draw a rectangle. Each letter should produce a different shape.

(5.1.c) Try changing the code by adding another condition of your own, when another key is pressed (other than R, G, B, or W).

5.2 Mouse Input

Below are the Processing variables and functions that handle mouse input. The functions are listeners and should be overridden if used.

- **mouseX** – x-coordinate of location in display window where mouse is
- **mouseY** – y-coordinate of location in display window where mouse is
- **mouseButton** – indicates if a mouse button is pressed: `LEFT`, `CENTER`, `RIGHT`
- **mousePressed** – boolean value set to `TRUE` if a mouse button was pressed
- **mouseClicked()** – function that is called when a mouse button is pressed and then released
- **mousePressed()** – function that is called when a mouse button is pressed
- **mouseReleased()** – function that is called when a mouse button is released
- **mouseMoved()** – function that is called when the mouse moves
- **mouseDragged()** – function that is called when the mouse moves while a button is pressed

Below is an example sketch that demonstrates mouse input:

```
int x, y;
void setup() {
  ellipseMode( CENTER );
  x = width/2;
  y = height/2;
}
void draw() {
  background( #ffffff );
  stroke( #ff0000 );
  strokeWeight( 2 );
  ellipse( x, y, 20, 20 );
}
void mousePressed() {
  x = mouseX;
  y = mouseY;
}
```

Enter this code in your text window and try it out.

Then:

(5.2.a) Modify the sketch so that the circle is only repositioned when the user drags the mouse.

(5.2.b) Modify the sketch so that the circle is repositioned whenever the user moves the mouse.

(5.2.c) Create a new sketch that draws the outlines of multiple shapes in the display window. Then if the user clicks the mouse inside a shape, the shape is redrawn and filled.

6 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.

Enter this sample code below into your text window. Try running it and watch what happens.

```
int x = 0;
int y = 50;
void setup() {
  background( #000000 );
}
void draw() {
  background( #000000 );
  ellipse( x, y, 40, 40 );
  x = x + 1;
  if ( x > width ) {
    x = 0;
  }
}
```

Then:

(6.a) Modify the code to make the circle move vertically instead of horizontally.

(6.b) Modify the code again to make the circle start with a small width and height and grow larger each time **draw()** is called. Decide what to do when the circle has grown to be too big to fit inside the display window—do you want it to shrink back down? or snap back to the original size? or something else?

7 Programming challenges

(7.a) Create a sketch that animates different shapes depending on different keys that the user presses. For example, pressing `C` might make a green circle float from left to right across the display window, and pressing `T` might make a blue triangle float from the top of the display window to the bottom. Have fun!

(7.b) Create a sketch that draws lines in the display window as the user drags the mouse. Change the color and width of the stroke based on keys the user presses. For example, the line turns blue if the user presses `B`; the line drawn is thick if the user presses `3`, or thin if the user presses `1`. Get creative!

8 On-line references

- Processing web site: `http://www.processing.org/`
- getting started tutorial: `http://www.processing.org/learning/gettingstarted/`
- reference: `http://www.processing.org/reference/index.html`