



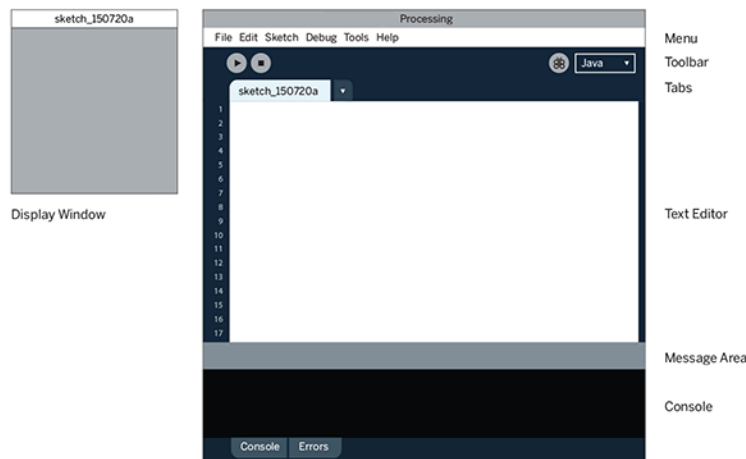
# Introduction to Programming using Processing

Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. There are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning and prototyping. Processing is available for free from <http://processing.org> for Windows, OSX, and Linux. You can also find Processing compatible apps for Android and iPhone in their app stores.

This introduction is based on the tutorials and guides available at <http://processing.org> and modified by Arch Reactor members for this workshop. Arch Reactor volunteers are here to assist you along the way. Don't hesitate to ask any questions!

Lets get started by launching the Processing IDE. IDE stands for Integrated Development Environment. Code development is done using multiple software tools, including a text editor to write code, a compiler to turn the code into something a computer can run, a debugger to help us find errors, and more. An IDE combines these tools into one software package specializes towards whatever programming language you are using.

You'll find the Processing icon on the desktop. If you're on one of our Linux based systems it will look like  and on the Windows based systems it will look like . It'll take a moment to fully startup, Once running it should looks similar to this:



The Display Window will only show when your Sketch, or processing program, is running. The Play and Stop buttons, like you might find on a DVD player, will start and stop your sketch, so when we say “run your code” press the play button. The white area is where you’ll write your code, and the black area is where you can see errors and use the Console. We’ll talk more about these things later as we need to use them.

Start with a very simple sketch, just one line. Enter the following into the text area:  
`ellipse(50, 50, 80, 80);`

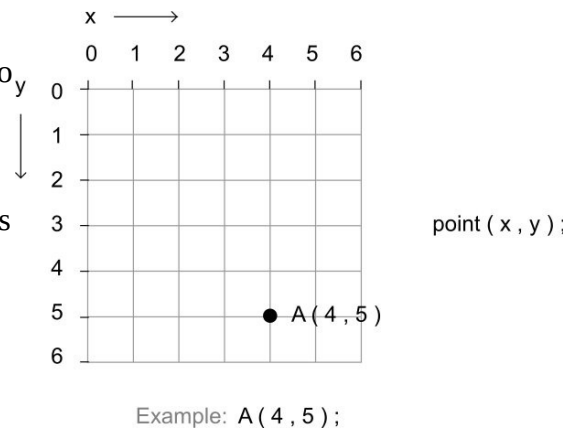
Now run your code! If you made any typing mistakes you will see a message about it in the Console area. Sometimes these messages aren't easy to decipher, so ask an Arch Reactor volunteer for help if you need to. At home you can paste the message into Google and likely find someone that had the issue before and a solution to it.

If you don't get any errors the display window should appear with a circle on it, which is an ellipse with equal width and height. You can close the display window, or press the stop button, to stop your sketch.

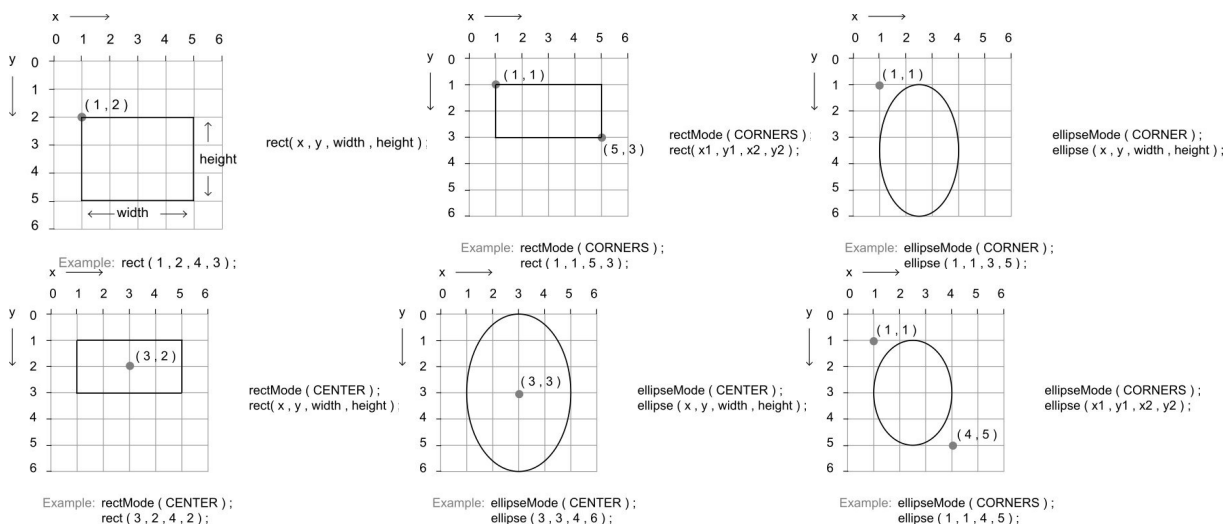
`ellipse()` is called a function, it's a command that runs some already written code to do a thing. Functions can take parameters inside the parenthesis to have the function adjust what it does. To understand how this `ellipse` function works, let's first look at the coordinate system. A display is made up of a whole lot of points laid out in a grid. You might have seen that a computer LCD has a "native resolution" of 1920x1080 or something similar. That means it has 1,920 points across and 1,080 points down in its grid. Processing uses this same type of system in its display window so that you can specify where to place your drawings.

This coordinate system is illustrated to the right using Processing's `point` command. Note that the numbering is zero based, so the 1st column is number 0 and the 5th column is number 4. This can take a little bit to get used to, but it's pretty handy to have it this way when doing many types of programming operations. Here are some more of Processing's drawing functions

```
point(x, y)
line(x1, y1, x2, y2)
rect(x, y, width, height)
ellipse(x1, y1, x2, y2)
```



`rect` and `ellipse` also have `CENTER`, `CORNER`, and `CORNERS` modes which are set using the `rectMode` and `ellipseMode` functions. `rect` defaults `CORNER`, `ellipse` defaults `CENTER`. Code in all caps like this are what we call constants. They have a special value in the program. Below are some examples of how this works on the grid. Go ahead and modify your `ellipse` code to help you understand how this all works.



If you have doubt at all about how the coordinate system works please ask for more explanation from an Arch Reactor volunteer now as this is crucial to getting Processing to draw things where you want them.

Lets move on to a simple program that we'll expand on throughout the guide. Delete any code you may have in the text area and enter the code to the right. We have a new function, `size`, that we can use to set the size of our display window.

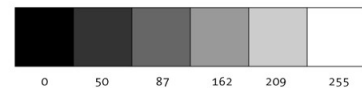
```
size(200,200);
rectMode(CENTER);
rect(100,100,20,100);
ellipse(100,70,60,60);
ellipse(81,70,16,32);
ellipse(119,70,16,32);
line(90,150,80,160);
line(110,150,120,160);
```

Run your sketch and see what this draws. Code is basically just a list of instructions that get carried out one at a time in the order listed.

The computer is fast enough that it seems to all happen at once.

Can you match up what line of code draws which part of the drawing? Lets add some color to make that task easier. There are 2 main ways we can have Processing give us color: Grayscale, and Red-Green-Blue, or RGB, colors. There's also a way to set a shape's transparency.

Grayscale is a single number from 0 to 255 for 256 total shades. 0=black, 255=white.



## RGB

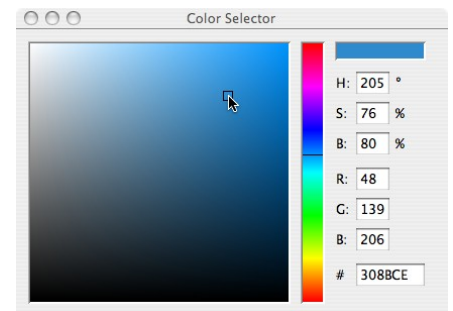
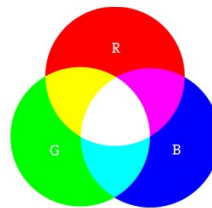
0=none of that color

255=all of that color

## Transparency

0=completely transparent

255=completely opaque



Processing also has an interactive Color Selector and HTML and HSB color code converter. You can find it in the Tools menu.

The 3 most common function in Processing to set colors are

`background()`;

`stroke()`;

`fill()`;

In all 3 you can use a single parameter for the grayscale or 3 parameters for the RGB code. For example, to have lines as the blue shown in the color picker above, you'd write `stroke(48, 139, 206)`; `stroke` and `fill` will apply to everything drawn after them until you set a different color.

Lets make our background black, lines white, the rectangle blue, one ellipse white and everything else green. Adjust your code to look like the code to the right. Run it see if you were right about which ellipse the first one is! You can adjust the colors and add more stroke and fill functions to color your design however you like.

```
size(200,200);
stroke(255);
background(0);
fill(0,0,255);
rectMode(CENTER);
rect(100,100,20,100);
fill(255,255,255);
ellipse(100,70,60,60);
fill(0,0,255);
ellipse(81,70,16,32);
ellipse(119,70,16,32);
line(90,150,80,160);
line(110,150,120,160);
```

So that's how to draw, but to make something interactive we need to create our own functions for Processing to use, specifically, the `setup()` and `draw()` functions. Some of our code will go in `setup` that runs once when the sketch starts, and the rest will go in `draw` that will get call anytime Processing wants to redraw the display window, which we can also control using the `frameRate()` function to specify how many times per second to draw the display window.

To make a function, we place curly brackets {} around our code, and add the function name and type before it. A type is our way to tell the compiler the best way to store it in memory and what kinds of things we can do to it. For `setup` and `draw` the type is `void` because it just runs and that's it. Adjust your code to be like that to the right and run it. Notice there is no drawing functions outside the {}, and our `background` is in the draw function. `background` doubles as a way to erase the screen, which will be important in a moment.

Once you have that working, it should look the same as before. Now we're ready to add some interactivity!

Mouse and Keyboard interactivity can be done using some variables that Processing gives us. A variable is a named value, kind of like the constants we talked about earlier, but the value can change as the program runs. The 4 variables for the mouse are `mouseX`, and `mouseY` to give us the point where the mouse is, and `mousePressed`, and `mouseButton` to tell us if we're clicking and which button using the constants `LEFT`, `CENTER`, or `RIGHT`. Let's replace a leg! Our sketch is getting a bit long to keep repeating here, so remove the last line, "`line(110, 150, 120, 160);`", then add the code to the right at the end of the draw function, before the last }.

`if()` is what's known as a "flow control" statement. It lets us choose which section of code to run based on a condition that we put inside its (). The sections of code are wrapped in {} just like a function. Since `mousePressed` is either true or false we don't have to compare it to anything, but to check if `mouseButton` is the left one we need to use the comparison operator `==`. The `&&` is "and" logic, which means that both the mouse has to be pressed and it has to be the left button for the whole condition to be true. If the condition is true, it runs the first code section, if it's not it runs the code section after the "else". The else section is optional if you don't any code you want to run. There is also "or" logic using `||` which is 2 pipes (shift backslash, just above Enter on most keyboards).

Processing can also draw text on the screen using the functions `text()` and `textSize()`. To control the text color use the `fill` function before it. Add this code at the end of the draw function, before the last }

```
void setup() {
  size(200,200);
  stroke(255);
  frameRate(30);
}

void draw() {
  background(0);
  fill(0,0,255);
  rectMode(CENTER);
  rect(100,100,20,100);
  fill(255,255,255);
  ellipse(100,70,60,60);
  fill(0,0,255);
  ellipse(81,70,16,32);
  ellipse(119,70,16,32);
  line(90,150,80,160);
  line(110,150,120,160);
}

if(mousePressed && mouseButton == LEFT){
  line(110,150,mouseX,mouseY);
} else {
  line(110,150,120,160);
}

fill(255);
textSize(16);
text("arms?", mouseX,mouseY);
```

Now see if you can place some arms on your own! You may use any time remaining to experiment with your sketch, make a new drawing, experiment with Processing's examples, or switch to the Sparki robot guide.