UNIT 2. PROCESSING AND PROGRAMMING BASICS

Summary

In this section, you will…

- Get and install the Processing program which enables you to write Processing computer programs
- Familiarize yourself with the Processing program, and the processing canvas
- Learn basic “syntax”, the rules of writing programs
- Write your very first program, and, encounter your first errors
- Learn about drawing and painting order, and how to change the color of what is drawn

Learning Objectives

After finishing this unit, you will be able to write basic Processing programs that draw simple shapes on a drawing canvas. Specifically, you will be able to:

- Choose and set the canvas size for your programs.
- Set the location where shapes are drawn on the canvas.
- Set the background, outline, and fill colors of your canvas and shapes.
- Draw lines, circles and ellipses, triangles, and points.
- Insert in-line and block comments to describe your program.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

2.1 **Introduction**

Processing (processing.org) is a Java-derivative language designed to enable programmers to quickly and easily learn how to do really cool stuff, mostly through computer graphics. While it is often seen as a teaching language, professionals use it as well for small projects and prototyping. Processing is Java under the hood, so it can grow with you as you become an expert. It also has full 3D graphics built in so you can build some really cutting-edge graphical programs with it.

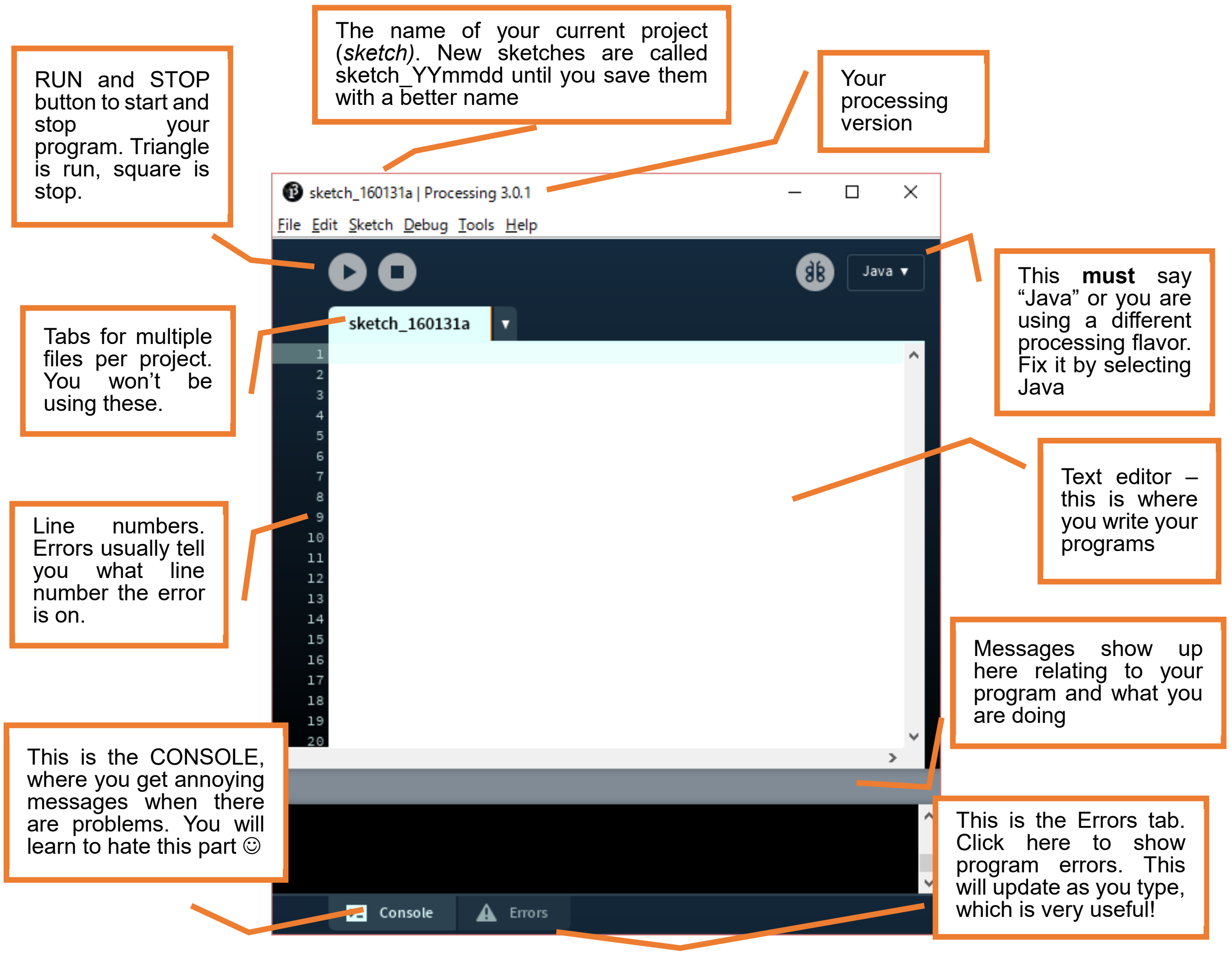
Processing has less testing with Window 8, and you should probably download the 32 bit version, which has been reported to be more stable.

You need to download Processing onto your own computer. Processing is free, and easy to install (ask the help center or the instructor if you're stuck). To start, go to http://processing.org/download and grab the latest version 3. The older versions will work but there will be slight differences in how it looks, so don't bother with that. Also, there is an installation guide for Windows, Mac, or Linux, which can help you to get it running, at https://processing.org/tutorials/gettingstarted/. Once you have it installed, you just double click on the `processing.exe` file and it will start to run.

Once started, you will see the Processing Development Environment. This manages all the under-the-hood parts of computer programming – we'll talk a little about that later in the course. For now, this enables you to learn to write computer programs without worrying about the nitty-gritty details of how the computer goes from what you type, to a program that runs and starts up. This is a good thing.

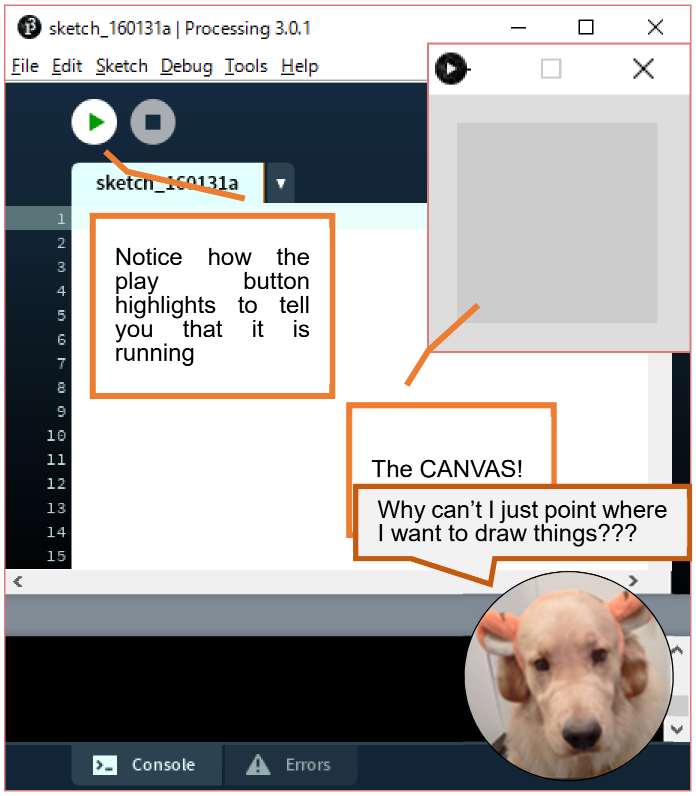
Processing does a lot of advanced things under the hood. It converts your program to JAVA, then converts that into a language called Java Byte Code, runs a Java Virtual Machine program, and runs your program with that.

Let's take a closer look at the processing development environment.


RUN and STOP button to start and stop your program. Triangle is run, square is stop.
The name of your current project (*sketch*). New sketches are called sketch_YYmmdd until you save them with a better name
Your processing version
This **must** say "Java" or you are using a different processing flavor. Fix it by selecting Java
Tabs for multiple files per project. You won't be using these.
Text editor – this is where you write your programs
Line numbers. Errors usually tell you what line number the error is on.
Messages show up here relating to your program and what you are doing
This is the CONSOLE, where you get annoying messages when there are problems. You will learn to hate this part ☺
This is the Errors tab. Click here to show program errors. This will update as you type, which is very useful!
sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
Java
sketch_160131a
Console
Errors

As the default program name suggests, Processing calls their projects *sketches*. This is fitting, as you get to quickly sketch up your ideas roughly to see how they work. Processing stores your sketches in a *sketchbook*, which is just a folder on your computer.

You can change where these are stored (e.g., ***to a backed up location!!!***) by looking at the `File→Preferences` menu dialog. All your sketches in your sketchbook can also be seen by going to `File->Sketchbook`. Discussing backup options is beyond the scope of this course, but it is highly recommended to have all of your work automatically backed up using one of the many free services currently available. Your University has a plan for a large amount of free storage. In addition, popular options include Dropbox, Microsoft One Drive, or Apple iCloud. Your instructor is unlikely to have patience for problems relating to a crashed hard drive or lost laptop. ***Important:*** Processing saves new sketches in a temporary location until you specifically save it to your own location, by clicking `File->Save`. Be sure to do this immediately when starting a new project so that you do not lose your work if your computer crashes.

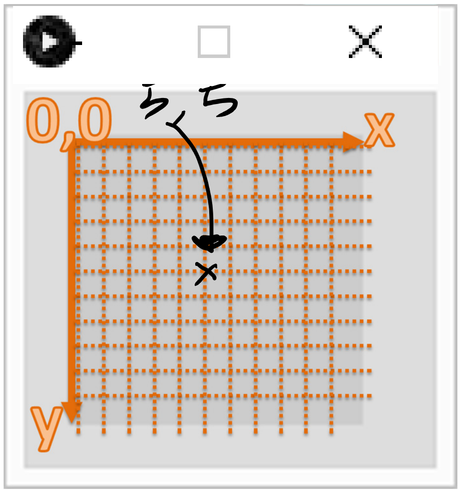

sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
sketch_160131a
Notice how the play button highlights to tell you that it is running
The CANVAS!
Why can't I just point where I want to draw things???
Console
Errors

2.2 The Processing Canvas

There is one more element to the Processing environment to introduce: the *Canvas* (officially called the *Display Window*, but I think Canvas is a better name). If you press *play* in Processing (even without any program typed up), an additional window pops up: the Canvas! You can press *stop* to get rid of the canvas and stop it from running.

Right now, the canvas looks very boring, but – just like a painting canvas – this is where you can be creative and start drawing whatever you want. Unfortunately, you cannot just paint by dragging the Mouse, but you can do it with programming instead!

Notice that the canvas has two sections. The center square, and the outside grey area. The center square is actually the canvas, where drawing happens. The outside is just "padding" to make it fit in the window.



Now, if you want to draw something on the canvas, you need a way to specify *where* to draw things. I suppose you could simply point with your finger or mouse. However, this limits you to placing things once. What if you want to animate something, like, to make a ball move across the screen? And what if this is hard to predict, like in a video game? The position keeps changing, and we need to be able to set it in our program. Because of this, the canvas uses a simple 2D coordinate system (Cartesian coordinates in Euclidean space), like you used in high school for graphing functions. There is an `x` axis, which goes from left to right, and a `Y` axis, which goes from top to bottom. The top-left corner of the screen is `(0,0)`. There are no negative coordinates. Notice the important difference when compared to high-school math: the `Y` coordinate increases as you go down. In regular math, it increases as you go up. Unfortunately, that is due to an old computer standard and is something you will have to remember.

The unit of measure is the pixel, which is the smallest unit of display on your screen. In processing, the canvas default size is 100 by 100 pixels. Don't worry – we will learn how to make a bigger canvas soon. Also, if you draw off the canvas it's no problem, processing just stops at the edge and continues your program.

2.3 Basic Processing Syntax: a command

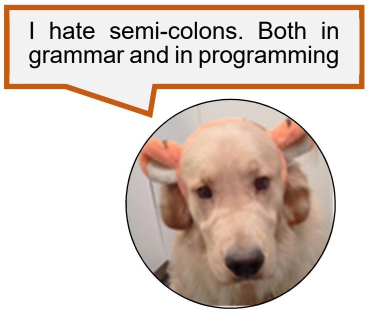
Computers are stupid. To give commands to a computer, we have to follow very strict rules of how to form the command. In programming, these rules are called the ***syntax*** of a language. Even if your command looks reasonable to you, if you do not follow the syntax rules, your program will not work. Think of syntax like grammar, except that if you make even the simplest grammar mistake, the computer has no clue what you tried to say. Throughout this course you will learn a lot of different syntax rules. Let's start with the simple command. Let's learn how to tell processing to draw a line on the canvas.

In order to give a command to a computer, you need to specify what command (e.g., draw a line on the canvas), and you need to give specifics to the command (e.g., where to draw the line on the canvas) – these are called parameters to the command.

Here is the basic syntax for a processing command:

```
command(parameter1, parameter2,...);
```

You give the name of the command, followed by an opening parenthesis (the regular round brackets). Then you add some parameters, with commas between them. You don't put a comma after the last parameter. Then you add a closing parenthesis, and you finish with a semi-colon. The semi-colon means "end of command". You can also add spaces here around the parenthesis and commas, but the formatting is generally like in the example above.



2.4 Your First Processing Command – Draw A Line

Type the following single line of code into the Processing development environment:

```
line(0,0,100,100);
```

and press run. You should see a line being drawn from the top-left corner of the canvas `(0,0)` to the bottom right corner `(100,100)`. Didn't work? Try the following...

- Did you remember the semicolon at the end?
- Did you use the correct brackets? The ( ) and not [ ] or { } or 「」 ?
- Did you type line and not LINE or Line? It is case sensitive (see below)…

Remember that the canvas defaults to being 100 by 100 pixels, and the top left corner is coordinate `(0,0)`. So, the `line` command takes four parameters: the first two are the `x` and `y` coordinate where the line starts, and the second two are the `x` and `y` coordinate of where the line ends. You can imagine the command is like this:

```
line(x1,y1,x2,y2);
```

and you can plug in whatever values you want. Try it, try drawing the line from and to different spots on the canvas.

So, how do you know which parameters (the stuff in brackets after a command) to use and what they mean for a command? Basically, it's confusing, and every command is different.

When is the last time you used the "`Help`" menu in a program? Never? Did you notice that many programs got rid of the help menu (like MS Office)? Well, programmers use on-line help all the time. Reference manuals are very useful for programming because it is not reasonable to expect people to remember all these details.

Having a perfect memory helps intensely!

Open Processing, and click `Help→Reference`. It should open a reference manual in your primary browser. Look for the `line()` command (hint: it's under 2D Primitives) and click on it. At first, this page may look overwhelming, but you will learn to love this reference manual. Notice that it has examples of how to use the command. It has a detailed description, much of which you will not yet understand. That is okay, you should be comfortable with peeking at it and grabbing what you understand. And it even has a Syntax section that explains how to use the command, with explanations of what the parameters are. Take a minute to read the page.

2.5 **Your very first Syntax Errors**

No one avoids mistakes in their syntax (syntax errors). The world's best and most experienced programmers (yes, even the dog!) have regular syntax errors. Syntax errors are not only annoying because you need to know the rules, but our stupid computers are actually really bad at explaining what

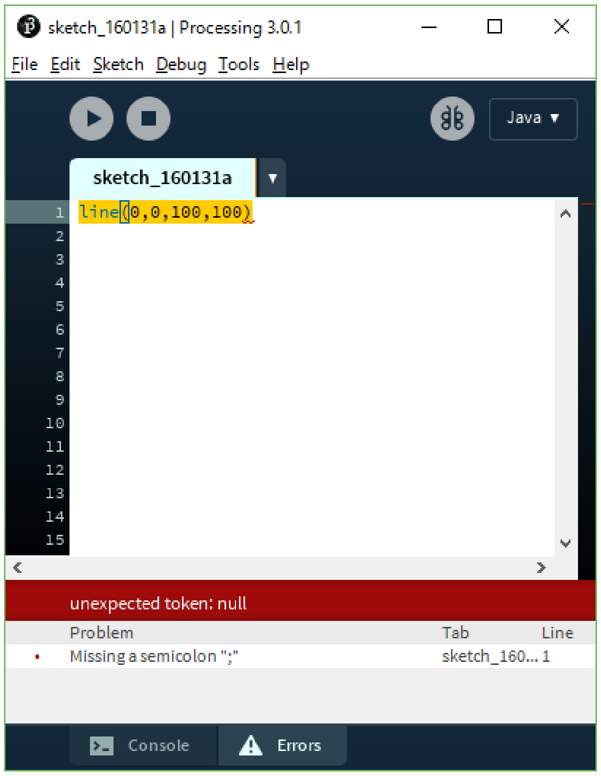
Syntax Errors are *compile-time* errors. This means that Processing refuses to even convert your program into computer language (binary). The error happens before it has a chance to run!

is wrong with your syntax. To make things worse, for all its awesomeness, Processing is particularly bad at explaining the syntax errors. Try this, type the following command in Processing – this is the command from the previous section, but we removed the semi-colon

```
line(0,0,100,100)
```

When you click *run*, nothing happens. If you look down on the Processing window near the bottom, you will see in the message bar that it says `unexpected token: null`. Now, I'm not sure what engineer thought that this is a nice, descriptive error message, but it's what we're stuck with. In this case, what it's really trying to say is: I expected something here (ahem, a semi colon), but I found nothing!! (`null`). One nice feature is that processing moves the cursor to the spot where it thinks the error is (the cursor is that vertical blinking bar that tells you where you are typing).

Processing 3 has a great new feature that tries to help you out with common errors. ***If you click on the "Errors" tab at the bottom of the processing window, you should see a list of errors.*** Here, it suggests that your problem is that you are missing a semi colon, and it even tells you which line (line 1 in this case). If you click on the error, it will even take you to that spot in your program, which is useful as your programs get large.


sketch_160131a | Processing 3.0.1
File Edit Sketch Debug Tools Help
Java ▼
sketch_160131a
Console
Errors

```
line(0,0,100,100)
```

```
unexpected token: null
Problem                    Tab             Line
Missing a semicolon ";"    sketch_160...   1
```

Fix the syntax error by adding a semi-colon. Your program should work now.

Let's try another one:

```
LiNe(0,0,100,100);
```

Now, Processing gives the syntax error: `The function LiNe(int, int, int, int) does not exist`. We'll learn about those `int`s later, but basically Processing doesn't understand the command `LiNe`. ***Processing (like many languages), is what we call case-sensitive***. This means that it sees upper case and lower case letters as completely different letters. The command `line` will *only* work if all letters are lower case. You'll catch yourself making this error throughout the course. So! If Processing says something doesn't exist, make sure you spelt it correctly AND are using the correct upper and lower case letters.

2.6 More Processing Commands

We need to improve our measly tiny canvas! Biggest canvas is best canvas, right? Whip out our Processing reference (remember, `Help`→`Reference`) and look up the `size()` command. (It's under Environment). Take a quick look through here. Notice how much of it will be over your head (whoa!! Rotate! 3D! what is a renderer?). That's OK – it's a reference, after all, and will include advanced stuff. What you need to home in on is the simple examples and the syntax. The syntax section says:

```
size(w,h);
```

where w is the width of the display in pixels, and h is the height. So, try placing the following command in a clean processing window (if you have stuff in there already, erase it first)

```
size(500,500);
```

and press *play*. You'll now have a nice big canvas! You can even make it bigger! Leave this **at the top of your programs** and you will always have a bigger canvas.

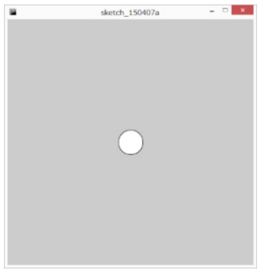
Another great command is `ellipse` – which lets you draw ellipses and circles. Remember, a circle is just an ellipse with the same width and height. Take a peek at the reference page again for `ellipse`, under the 2D Primitives section, and you'll find the following syntax entry (no seriously, go look. This is good practice).

```
ellipse(a, b, c, d);
```

where a and b are the x and y of the ellipse center, and c is the width, and d is the height (`a`,`b`,`c`, and `d` are kind of stupid names for those parameters!!)

So let's draw a circle of diameter 50 at the center of the screen. Remember that our size is now 500 by 500, so the center is `(250,250)`. Erase your processing program and copy the following into processing:

```
size(500,500);
ellipse(250,250,50,50);
```



If all goes well, you should see the canvas to the right with a single circle in the center. Try playing with the parameters of the commands to see what kinds of results you can come up with!

Check your Understanding

2.7 **Check your Understanding Part 1 – a quick break!**

Usually exercises come at the end of a unit. However, at this stage you need to do some work on your own.

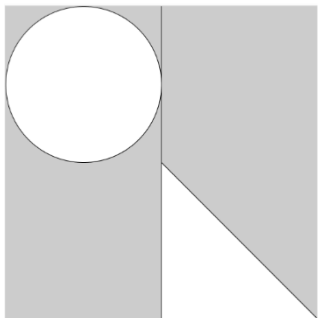
CuSn

Exercise 1. Investigate the following commands on your own:

a. `point` . This draws a single point. Make a program that uses this command at least three times. This is not a trick question – they will be VERY small and hard to see, so look closely.
b. `rect` . This draws a rectangle. Make a program that draws at least three rectangles.
c. `triangle` . As you may have guessed, this draws a triangle given three points. Make a program that draws at least three triangles.

Ag

Exercise 2. Using the commands you learned so far, create the sketch shown to the right. It has 1 line, 1 ellipse, and 1 triangle.

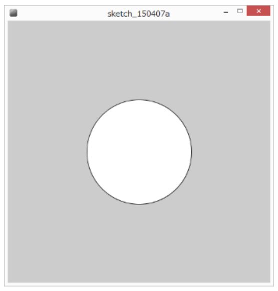


2.8 **Paint on top of Paint (order of commands)**

Did you know that the Mona Lisa was painted on top of a different portrait?

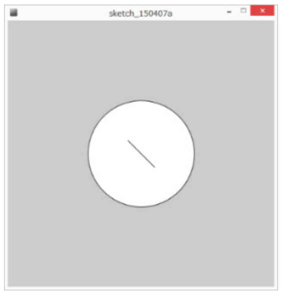
When you draw on a canvas, you can paint a nice picture and then, paint on top of it to hide whatever was underneath. Because of this painters need to plan ahead – paint the nice broad blue sky first maybe, then add some trees on top. Doing the trees first and then painting in the blue sky is much harder.

Processing has the same principle. In computer programming, operations are generally run from top to bottom. Later you will learn techniques to make it jump around like crazy, but for now, we start at the top and go down. Try typing and running the following code



```
size(500,500);
line(225,225,275,275);
ellipse(250,250,200,200);
```

You will see the image on the right. Where is the line? Processing didn't forget to draw it. It drew it first, since – top to bottom, the line command comes first – and then drew the circle on top of the line. Try the following variant changing the order of commands:



```
size(500,500);
ellipse(250,250,200,200);
line(225,225,275,275);
```

Now, you see the right image instead.

The order of the commands issued to the program will impact the result. This is not only the case when drawing images, but later, you will see also that this is the case when doing other operations like mathematics. Just remember – top to bottom, one command at a time.

2.9 **Organizing Bigger Scenes: Adding Comments**

Quick! What does the following program do?

```
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,40);
ellipse(325,225,60,40);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```

What do you mean you have no idea? Isn't it obvious? No? Why not? This is hard to understand because series of commands are hard to understand unless you are the one who wrote them. And even then, give it a few weeks, and you'll forget too. There are a lot of ways that programmers use to make their code more *readable* and understandable by people. One very important way is the addition of comments.

Comments are English language additions to programs that only serve the purpose of helping a person read the program. The computer completely ignores them.

One type of comment is the **block comment**. This lets you tell Processing that a whole region is text-for-humans and no computers are allowed inside! You can make a block comment by starting it with the `/*` symbols (forward slash then star) and ending with `*/` symbols (star then forward slash). For example

```
/* this is a comment */
```

These are called block comments because they can span many lines and make up a whole block:

```
/* this is a comment, too
   But it keeps on going and going.
   Eric the fish.
   Eric the fruit bat
   Eric the cat
   And don't forget Eric the kangaroo
*/
```

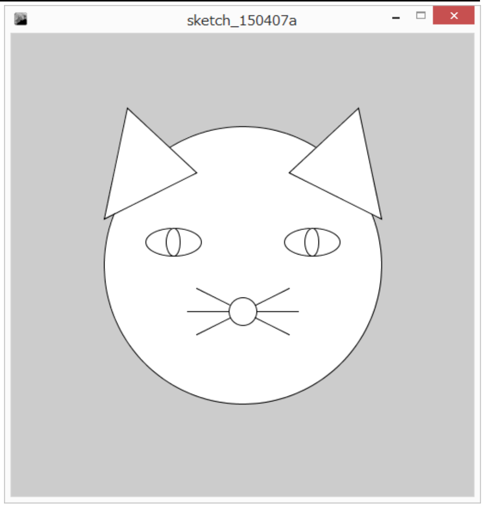
What if I want to put the */ symbols inside a comment? Will the computer know it's part of my comment or get confused, thinking I ended my comment?

Everything between that start and end character is ignored by the computer.

A very common block comment is the header of your program. At the beginning of a program it is common to give the key information about the program, such as who made it, what the purpose of the program is, etc. ***Header comments are required for all your assignments.***

Let's add a header comment to our previous program from the beginning of this section.

```
/********************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,30);
ellipse(325,225,60,30);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```



Now we understand what this program does. The graphical output is shown on the right:

Now, what if I asked you to change the size of the nose, or move the eyes. Which commands would you change? Which `ellipse` command draws each part? You may be good enough to eyeball it (hah!) and figure out which numbers correspond to which piece. But, there's an easier way. Inline comments.

Inline comments are comments that only go from where it starts until the end of that line. It's a simpler comment. You start them with `//` and everything after it on the line is ignored.

```
line(10,10,200,200); // this draws a diagonal line
```

We can add comments when our program is not clear, to help the reader (and ourselves!!) to understand what is going on.

Also notice how I use whitespace (extra lines) to group. Now check out our fully commented version of the program.

```
/********************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);

// draw the nose. draw after whiskers for nice overlap effect
ellipse(250,300,30,30);
```

Here is a comparison to the original version of the program:

```
size(500,500);
ellipse(250,250,300,300);
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);
ellipse(175,225,60,40);
ellipse(325,225,60,40);
ellipse(175,225,15,30);
ellipse(325,225,15,30);
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);
ellipse(250,300,30,30);
```

```
/*******************
* Cat Face! Draw a cat face on the screen
* author: Teo the dog
* version: try #awesome
* purpose: to show how a cat can be drawn
********************/

size(500,500); // make a 500x500 canvas

//draw the head
ellipse(250,250,300,300);

//draw the ears
triangle(375,80,300,150,400,200);
triangle(125,80,200,150,100,200);

//draw the eyes
ellipse(175,225,60,30); // left eye
ellipse(175,225,15,30);
ellipse(325,225,60,30); // right eye
ellipse(325,225,15,30);

//whiskers!
line(250,300,200,275);
line(250,300,300,275);
line(250,300,190,300);
line(250,300,310,300);
line(250,300,200,325);
line(250,300,300,325);

// draw the nose. draw after whiskers
for nice overlap effect
ellipse(250,300,30,30);
```

To the computer, the program is identical. But to a programmer, the commented version is much easier to read and work with. ***You will be required to properly comment your assignments.***

One last thing. When do you use a block comment, and when inline? It's basically up to you, but usually inline is for small comments and block for large.

2.10 **Choosing your Paint**

Processing is capable of full color, but for simplicity sake, in this course we generally stick to black and white – or more accurately, greyscale. Processing is capable of a full range from pitch black (well, as black as your screen) to full white. You set the color by specifying it by number. You can think about this number as how much *brightness*. So, a color of `0` is full black. White, however, is not `100` or some other reasonable number. The brightest color is actually `255`.

For Information Only (not testable): You don't really need to understand why, but for the curious, computers *love* powers of two. That's because when you store everything in binary (on and off switches), you end up hitting powers of two all the time. If you are curious about the math, Processing sets aside 8 on or off switches for the greyscale color – this is called an 8 bit number. If you have *n* switches lined up in a row, imagine light switches, then you end up have 2^n possible combinations of those switches. So with 8 switches, you get 2^8 = 256 combinations. 0...255 is 256 different numbers (include the 0!!).

You can set the paint brush color in processing with the `stroke` command.

```
stroke(gray);
```

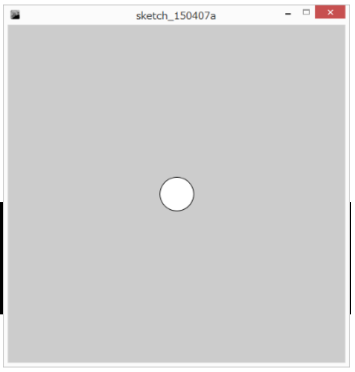
where gray equals the shade from 0...255.

Try the following program

```
stroke(0);
line(0,0,500,500);
```

and this slight variation

```
stroke(255);
line(0,0,500,500);
```



what is the difference? Try it out. You can see that the paint color changed.

Now, try the same comparison, but instead of drawing a line, draw an ellipse:

```
stroke(255); // or stroke(0)
ellipse(250,250,50,50);
```

Notice the difference? In the case with the black paint (color 0), only the outline of the circle changed color and not the center of the circle! So how would we get a black circle?

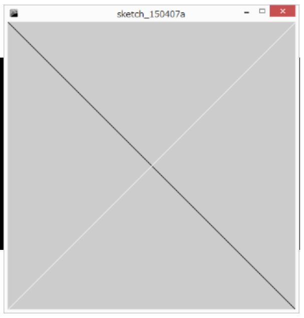
It turns out that processing has two different paints in use at any time. One is the stroke paint (the main brush for outlines). The other is the fill paint (what goes inside a shape).

```
fill(gray) // sets the fill color from 0..255
```

So, we can do a circle with a white outline and a black fill with the following commands.

```
stroke(255);
fill(0);
ellipse(250,250,50,50);
```

You can also switch out your paints at any time. Remember that Processing runs top to bottom. If you use `stroke` and `fill`, they keep fixed for all your drawing commands *until you change them*. For example, you can draw a black line followed by a white line as follows:

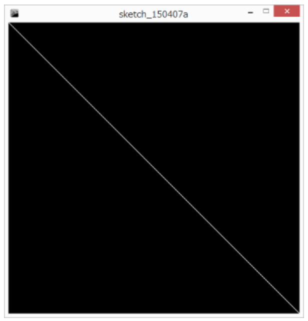


```
stroke(0); // set paint to black
line(0,0,500,500); // first diagonal
stroke(255); // set paint to white
line(500,0,0,500); // second diagonal.
```

There is one thing remaining. How do you reset that annoying ugly grey background? One way would be to draw a rectangle the size of the screen with the stroke and fill color of what you want. Luckily, Processing makes it easier: you can use the `background` command

```
background(gray) // erase the canvas and set to 0..255
```

For example:



```
background(0); // clear to black
stroke(255); // draw with white
line(0,0,500,500);
```

Now, you have all the tools you need to do the full range of colors!!

Check your Understanding

2.11 Check your Understanding: Exercises

Exercise 1.   What is syntax? In your own words, try explaining what syntax is to a person with no programming or computer background.

a. How much leeway do you have in your syntax? What happens if you have even a small deviation from the rules?

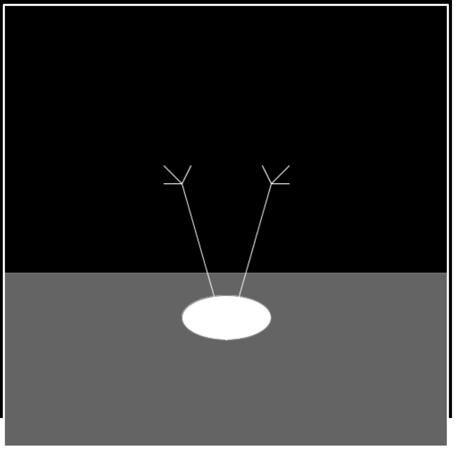
Exercise 2. Make a new Processing program with a 500x500 canvas. Draw a line from the top left corner to the bottom right corner.

a. Did you use 500 or 499 for your end coordinate? Why? Which is correct? (answer: 499).
b. If you have a canvas of size `n` by `n` pixels, what is the coordinate of the last pixel in terms of n?

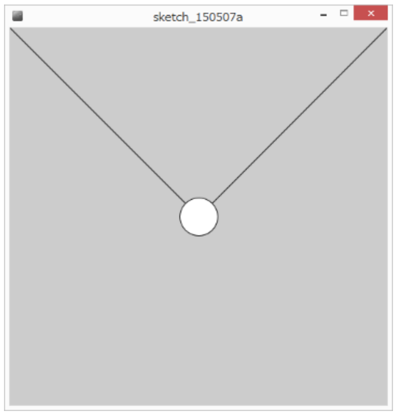
Exercise 3. Here is a program with a series of errors. One way to find the errors is to type the program into Processing and get its help to find them. However, in this case, look at the program yourself and see if you can spot all the errors first. Then, try typing it all up and see if you got them all. There are a lot, 23 in total! The final result should look like the inset at the end of the code.

```
/* broken program
   A bird who caught a very strong worm

canvas(500,500);
Background(0);
Strike(125);
fillColor(100);
rectangle(0,300,499);
fillColor(0);
Circle(250,350,100,50);
strike(256)
Line(250,375,200,200);
Line(200,200,180,180)
Line(200,200;180,200);
Line(200,200,210,180);
line(250,375,300,200);
line(300,200,320,180)
line(300,200,320,200);
line(300,200,290,180);
```

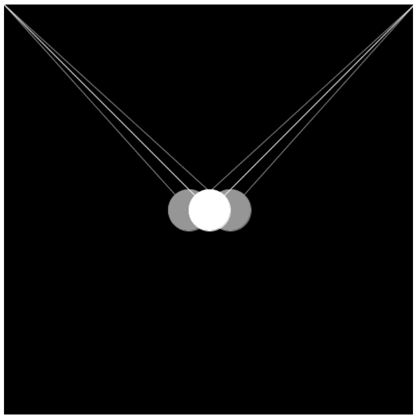


Exercise 4. Write a Processing program as specified below. If you forget how the commands are used, check your class notes for examples or use the reference in the `Help->Reference` menu



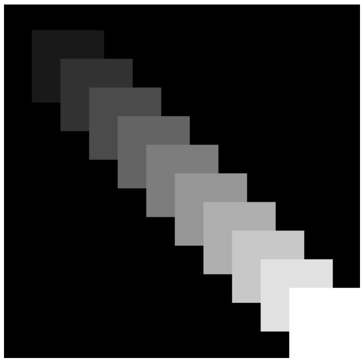
a. Make the canvas be 500 by 500 pixels large. (use the `size` command)
b. Draw a perfect circle at the center of the screen, with a diameter of 50 (use the `ellipse` command)
c. Draw a line from the center of the screen to the top left corner (use the `line` command)
d. Draw a line from the center of the screen to the top right corner
e. Make sure the circle is on top of the lines. You should get an image as shown.
f. Make sure you program has a block comment at the top describing your name, the course, and the purpose of the assignment. Also put at least one in-line comment.

Exercise 5. Write a Processing program as specified below. Make sure to have a block comment at the top and proper in-line comments. You will make the diagram as shown in the inset.



a. Set the canvas to be 500 square and clear the background to black
b. Set the stroke and fill colors to solid white, and draw a circle with radius 50 in the center of the screen.
c. Draw white lines from the top corners to the ellipse
d. Set the stroke and fill colors to 150 gray. Draw circles 25 pixels to the right and left of the center circle. Then, draw lines to those
e. You can try adding more circles and increasingly dark colors to create a bigger motion effect.

Exercise 6. Create a processing program to generate the image on the right. There are 11 squares (one is completely black!) so you should be able to calculate the square positions and colors using that knowledge. The square sizes can be whatever looks reasonable.



Exercise 7. Do a web search for “Droodle” – these are popular puzzles that use line drawings to depict a scene. At first, it looks abstract, but once you understand what it is showing, you can see the image.

Pick a few Droodles and try drawing them with processing.

Exercise 8. Read the Processing tutorial on color, which can be found at https://processing.org/tutorials/color/. While we will not use color in this course, color is not hard, and you can quickly learn how to use it. Try modifying some of the above examples to use color.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.

(page intentionally left blank)