

*Corso di Laurea Magistrale in Design, Comunicazione
Visiva e Multimediale - Sapienza Università di Roma*

Interaction Design

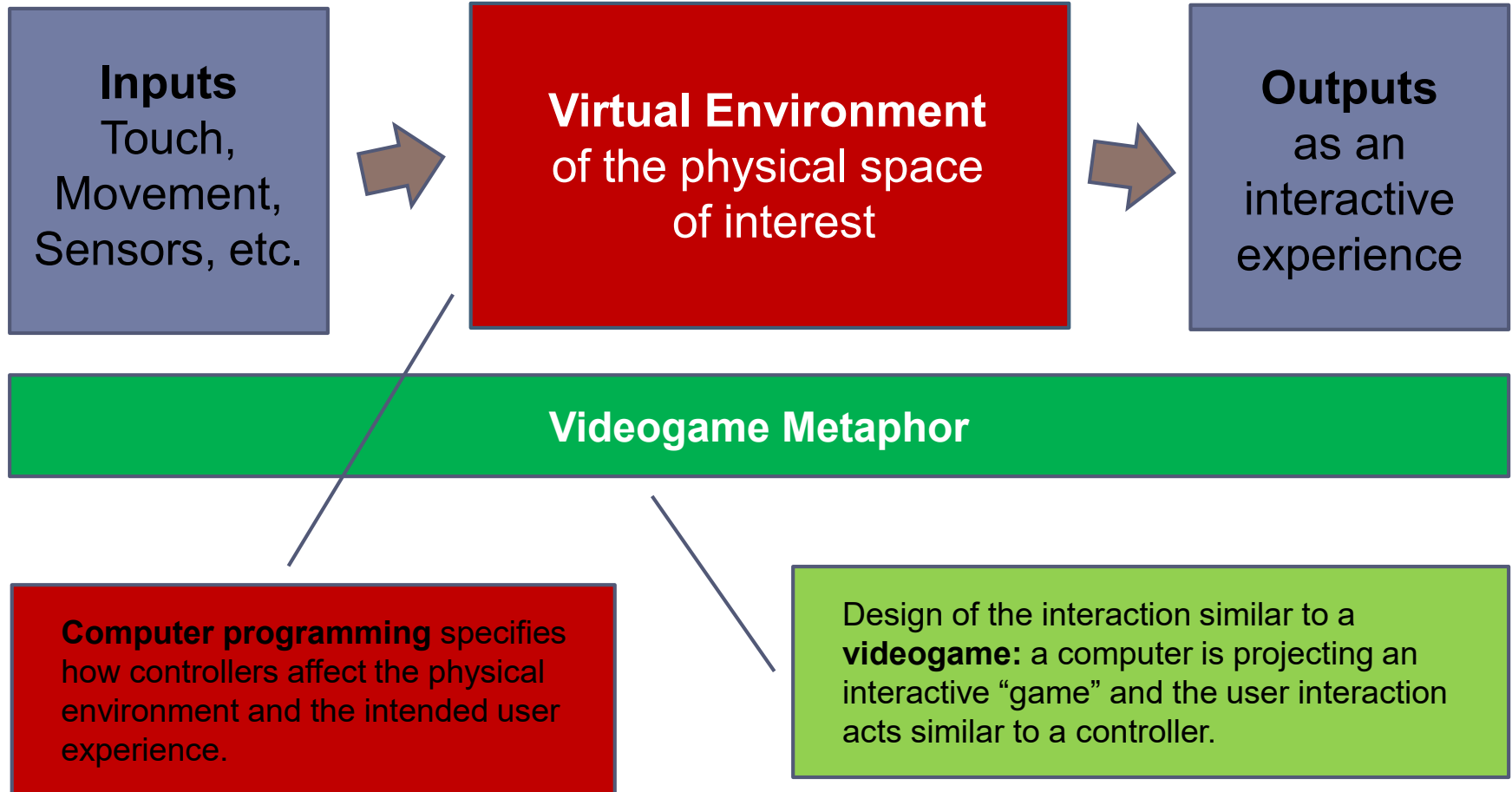
A.A. 2017/2018

5 – Basics of Processing

Francesco Leotta, Andrea Marrella

Last update : 12/4/2018

Ingredients for interacting with a smart environment

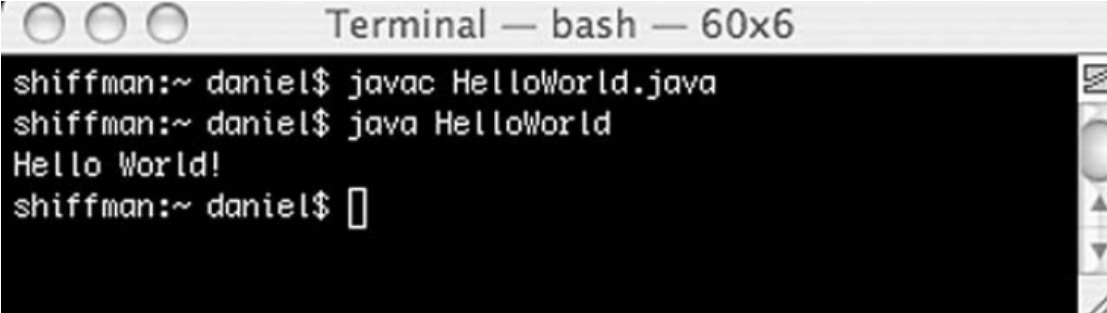


Computer Programming

- ▶ The purpose of **programming** is to find a **sequence of instructions** that will **automate performing a specific task** or **solving a given problem**.
- ▶ **Computer programming** is a process that leads to **executable computer programs**.
- ▶ We start by a **program** as a list of **statements** using the vocabulary of a **programming language** written as **simple text**, called **source code**.

Source Code

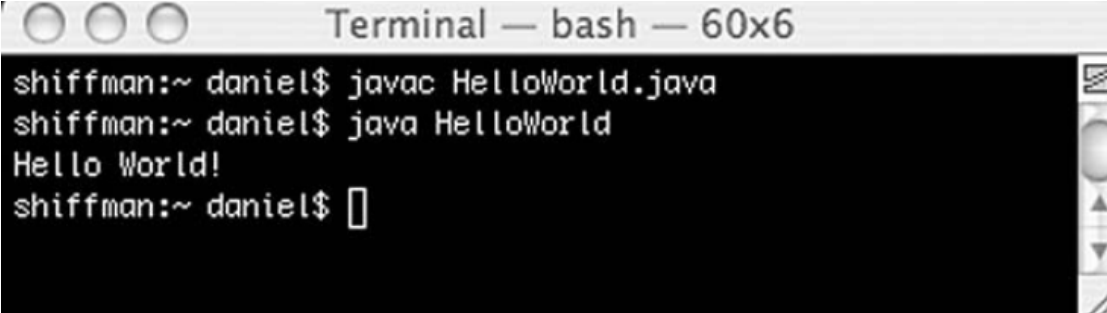
- ▶ **Source code** is any **collection of computer instructions**, possibly with comments, written using a human-readable **programming language**, usually as ordinary text.
- ▶ The source code of a program facilitates the work of computer programmers, who **specify the actions to be performed by a computer** mostly by writing source code.
- ▶ The source code is interpreted by the computer and thus **immediately executed**.



```
Terminal — bash — 60x6
shiffman:~ daniel$ javac HelloWorld.java
shiffman:~ daniel$ java HelloWorld
Hello World!
shiffman:~ daniel$
```

Source Code

- ▶ Source code is used to produce **an output** which may be a program like the ones you use occasionally.
 1. **TEXT IN** → You write your code as text.
 2. **TEXT OUT** → Your code produces text output on the command line.
 3. **TEXT INTERACTION** → The user can enter text on the command line to interact with the program.
- ▶ This example program returns a string **“Hello, World!”** in output.



```
Terminal — bash — 60x6
shiffman:~ daniel$ javac HelloWorld.java
shiffman:~ daniel$ java HelloWorld
Hello World!
shiffman:~ daniel$
```

Computer Programming

- ▶ A lot of programming languages:
 - ▶ C, C++, Java, Perl, Python, JavaScript, PHP, Ruby, ...
- ▶ A lot of terminology:
 - ▶ Variable, value, type, class, function, method, routine, interface, reference, array, conditional, loop, ...
- ▶ A lot of packages used on top of core languages:
 - ▶ Rails, Django, JQuery, OpenCV, ...
- ▶ The fundamental principles are **simple** and similar to all programming languages.
- ▶ You can use a lot of functionalities as “**Lego bricks**” as long as you understand how to put them together.

The Power of Computer Programming

- ▶ *“The programmers of tomorrow are the wizards of the future. You’re gonna look like you have magic powers”*

Gabe Newell, founder of Valve

<https://www.youtube.com/watch?v=nKlu9yen5nc>



Computer Programming

- ▶ We will look over some **basic things** to get you started with programming.
- ▶ There is an **enormous amount of information online**, there is always someone who had a similar challenge, and usually there is documentation for it.
- ▶ The intention of this course is to teach you programming using **Processing** as learning environment.
- ▶ The focus will be on the **core computational concepts** of Processing, which will carry you forward in your digital life as you explore other languages and environments.
- ▶ It is assumed that you have **no previous knowledge or experience of programming**.

Processing programming language

- ▶ Download the Processing language and programming environment from the following link:
 - ▶ <https://processing.org/download/>



Download Processing

https://processing.org/download/?processing

The screenshot shows the Processing.org website's download page for version 3.3. The page has a dark blue header with a network-like pattern and the word 'Processing' in white. A navigation bar at the top contains links for 'Processing', 'p5.js', 'Processing.py', 'Processing for Android', and 'Processing Foundation'. A search bar is located on the right side of the header. On the left, a vertical sidebar lists various site sections: Cover, Download, Exhibition, Reference, Libraries, Tools, Environment, Tutorials, Examples, Books, Handbook, Overview, and People. The main content area features a large heading 'Download Processing. Processing is available for Linux, Mac OS X, and Windows. Select your choice to download the software below.' Below this is a circular logo with the number '3' and the text '3.3 (12 February 2017)'. To the right of the logo are three columns of download links: 'Windows 64-bit' and 'Windows 32-bit' in blue; 'Linux 64-bit', 'Linux 32-bit', and 'Linux ARMv6hf' in blue; and 'Mac OS X' in a lighter blue. At the bottom, there are three links: '» Github', '» Report Bugs', and '» Wiki', followed by a paragraph: 'Read about the changes in 3.0. The list of revisions covers the differences between releases in detail.' and a final link '» Supported Platforms'.

Processing p5.js Processing.py Processing for Android Processing Foundation

Processing

Cover
Download
Exhibition
Reference
Libraries
Tools
Environment
Tutorials
Examples
Books
Handbook
Overview
People

Download Processing. Processing is available for Linux, Mac OS X, and Windows. Select your choice to download the software below.

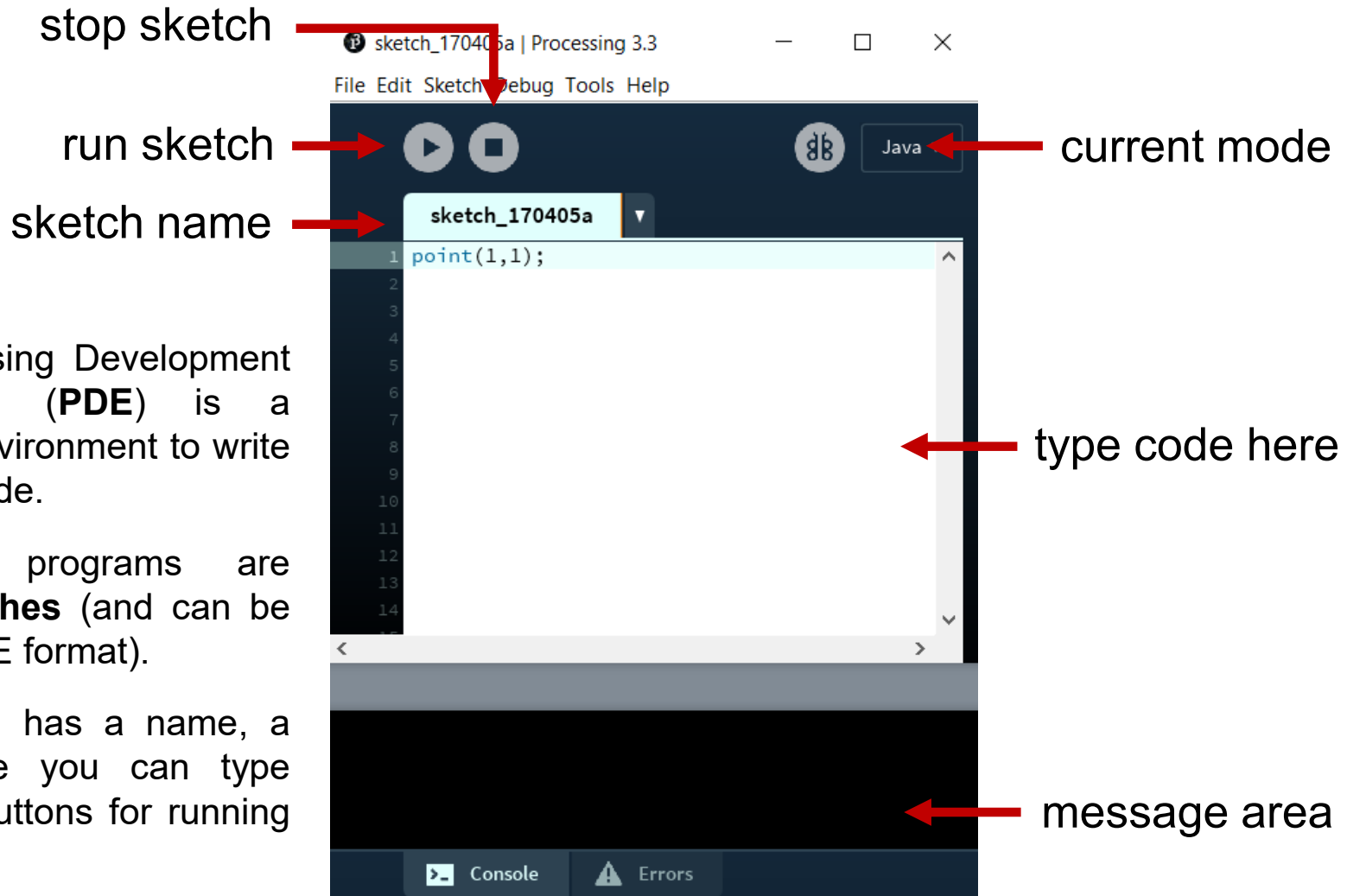
 3.3 (12 February 2017)

[Windows 64-bit](#) [Linux 64-bit](#) [Mac OS X](#)
[Windows 32-bit](#) [Linux 32-bit](#)
[Linux ARMv6hf](#)

» [Github](#)
» [Report Bugs](#)
» [Wiki](#)
» [Supported Platforms](#)

Read about the [changes in 3.0](#). The [list of revisions](#) covers the differences between releases in detail.

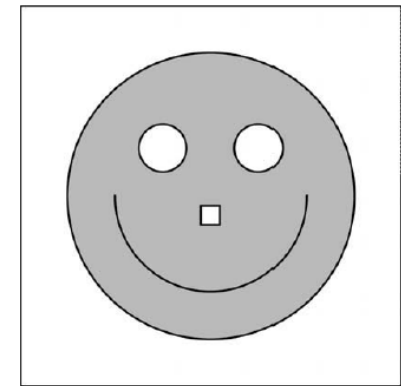
Processing Development Environment



- ▶ The Processing Development Environment (**PDE**) is a simplified environment to write computer code.
- ▶ Processing programs are called **sketches** (and can be saved in PDE format).
- ▶ Each sketch has a name, a place where you can type code, and buttons for running sketches.

Why Processing?

- ▶ It is **free** and **open source** (developed at the MIT Lab in 2001)
 - ▶ It is a **fully functional language** built on top of the Java programming language.
 - ▶ There is very little you can not do with Processing.
 - ▶ It provides a **more intuitive** and visually **responsive environment**, which is more conducive to **artists** and **designers** learning programming.
1. **TEXT IN** → You write your code as text.
 2. **VISUALS OUT** → Your code produces visual feedbacks in a window, to see what the code is doing.
 3. **MOUSE INTERACTION** → The user can interact with those visuals via the mouse.



Processing's "Hello, World!" might look something like this.

Why Learning Processing?

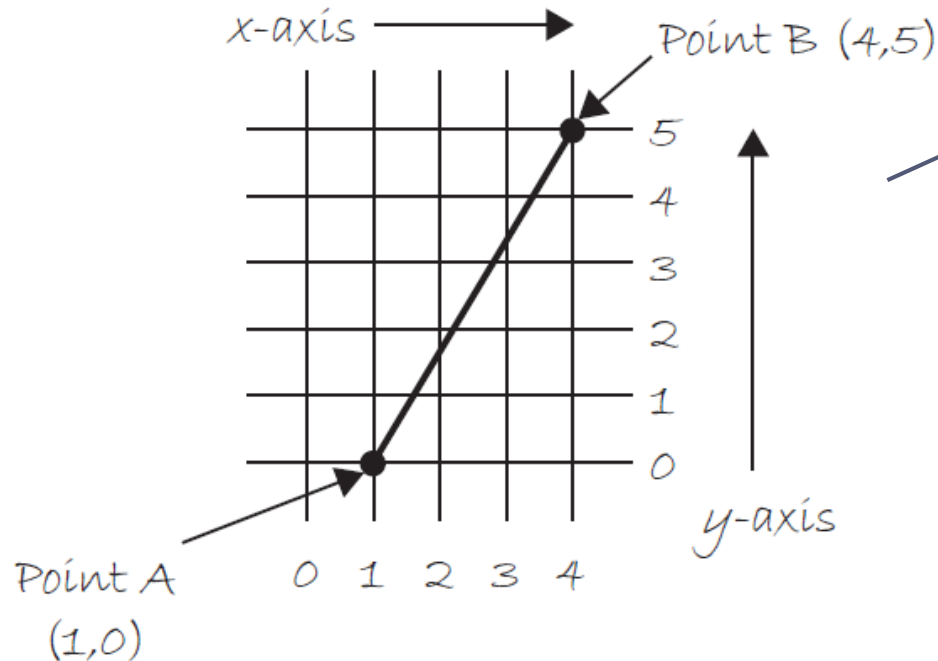
- ▶ *“Processing is especially **good for any person studying or working in a visual field**, such as graphic design, painting, sculpture, architecture, video, illustration, web design, etc.”.*
- ▶ *“If you are in one of these fields (at least one that involves using a computer), you are probably well versed in a particular software package, possibly more than one, such as Photoshop, Illustrator, AutoCAD, After Effects, and so on”.*

Daniel Shiffman, “Learning Processing”

- ▶ The main target is to **go beyond**, at least in part, from the confines of existing tools.
 - ▶ What can you make, what can you design if, instead of using someone else’s tools, you write your own?

Let's Start!

- ▶ Pull out a piece of graph paper, and draw a line.
- ▶ The **shortest distance between two points** is a good old fashioned **line**, and this is where we begin, with two points on that graph paper.



The figure shows a line between point A (1,0) and point B (4,5).

Let's Start!

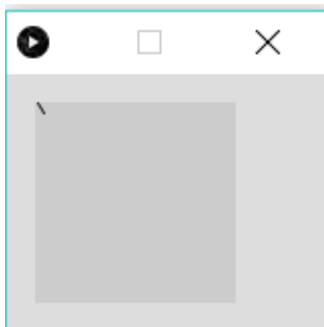
- ▶ Let's now imagine that we want to “***draw a line from the point one-zero to the point four-five***” with a computer and instruct it to display that same line on its screen.

- ▶ **How to perform this task through Processing?**

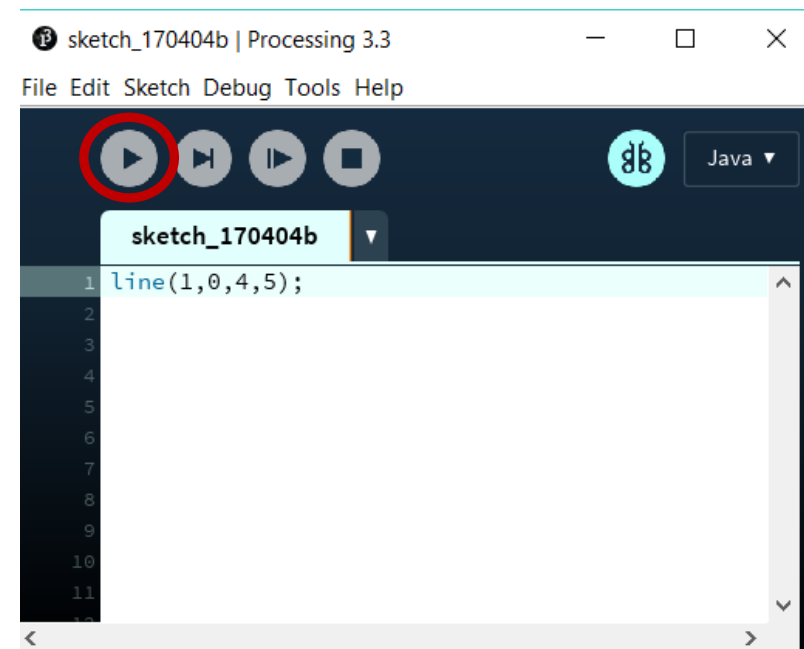
1. Open Processing
2. Write the following statement:

```
line(1, 0, 4, 5);
```

1. Press the Play button.

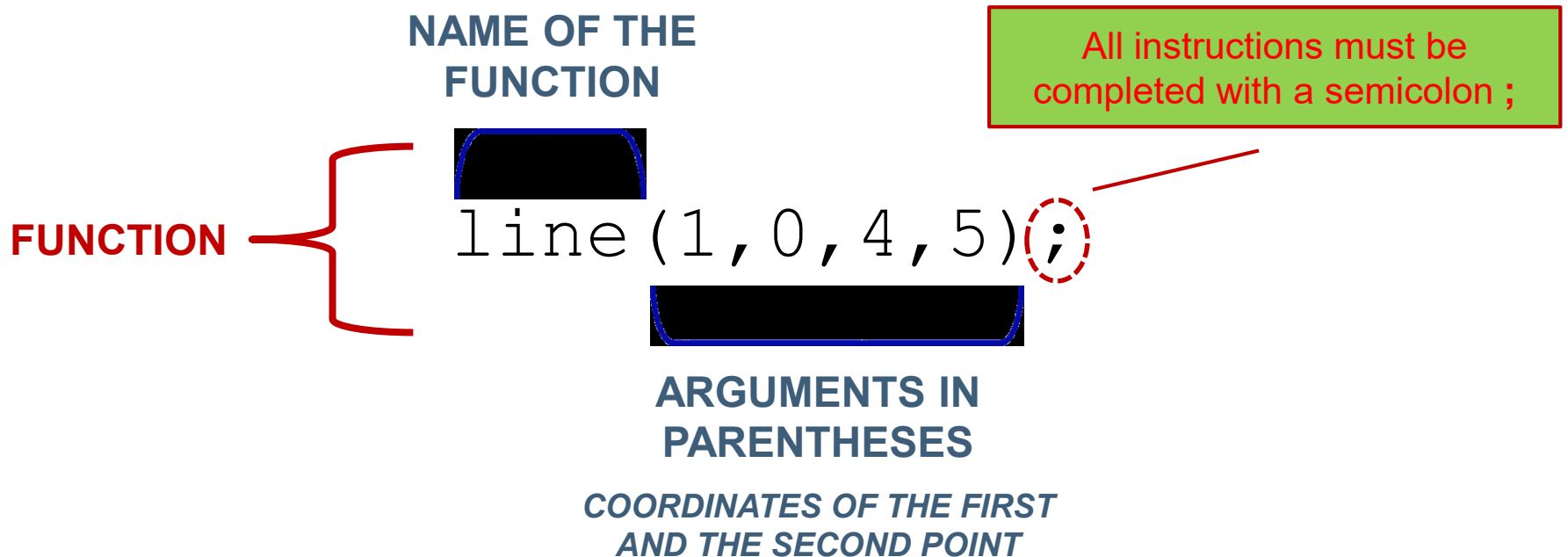


Congratulations, you have written your first line of computer code!



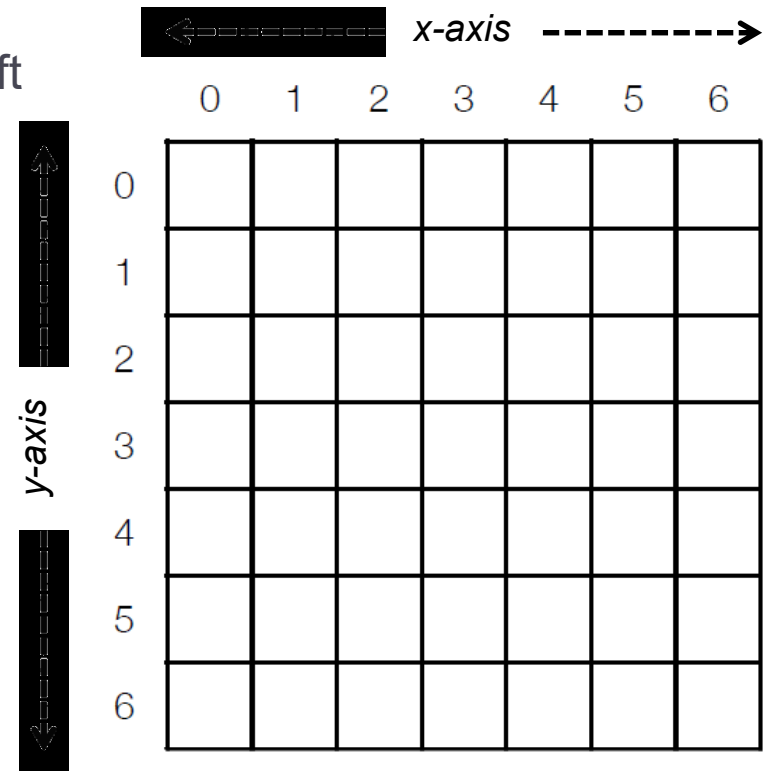
Our first function in Processing

- ▶ We are providing a command (we will refer to as a **function**) for the computer in order to draw a line.
- ▶ In addition, we are specifying some **arguments** for how that line should be drawn, from point A (0,1) to point B (4,5).



Coordinates

- ▶ The screen is like a piece of graph paper.
- ▶ Each cell is a **pixel** (“picture element”).
 - ▶ The origin (0, 0) is at the top left
 - ▶ x-axis: **+ is to the right**, - is to the left
 - ▶ y-axis: **+ is down**, - is up
- ▶ To draw objects with Processing it is required to instruct the computer with the **exact location** (pixels) where the shape starts to be drawn and its **size**.



Size of a window

- ▶ It is possible to specify the **dimension** of the window to be created through two arguments, **width** and **height**.

```
size(640, 480);
```

Open a window of width 640 pixels and height 480 pixels.

- ▶ In order to show a sketch fullscreen, use `fullScreen()` instead of `size(w, h)`

```
fullScreen();
```

Open a full screen window.

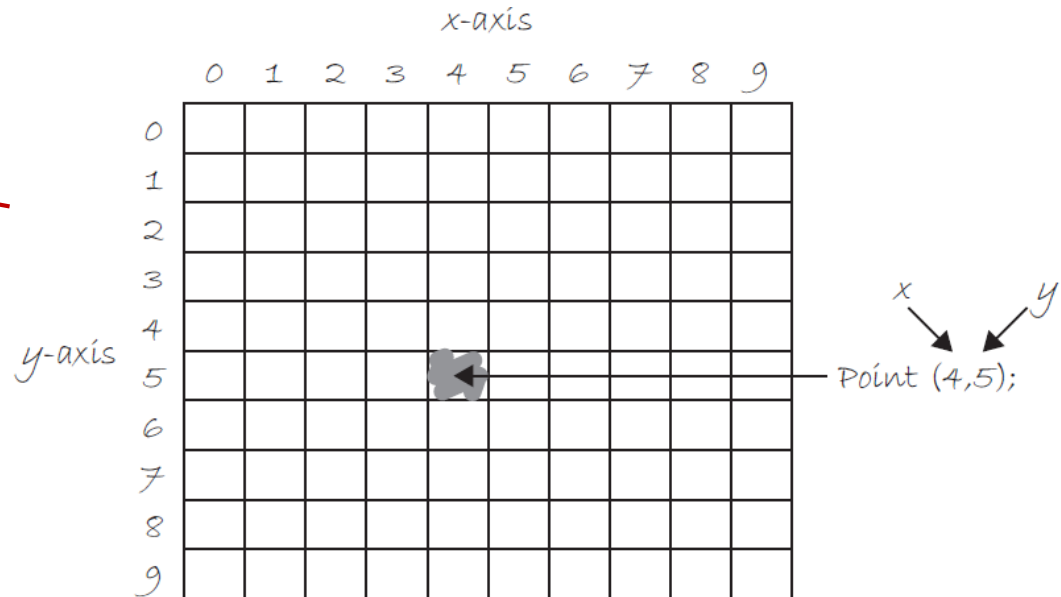
- ▶ The `size(w, h)` and `fullScreen()` functions can be used just **once** in a given sketch.

Points

- ▶ A point is the easiest of the shapes.
- ▶ To draw a point, we only need an x and y coordinate.

```
point (x, y) ;
```

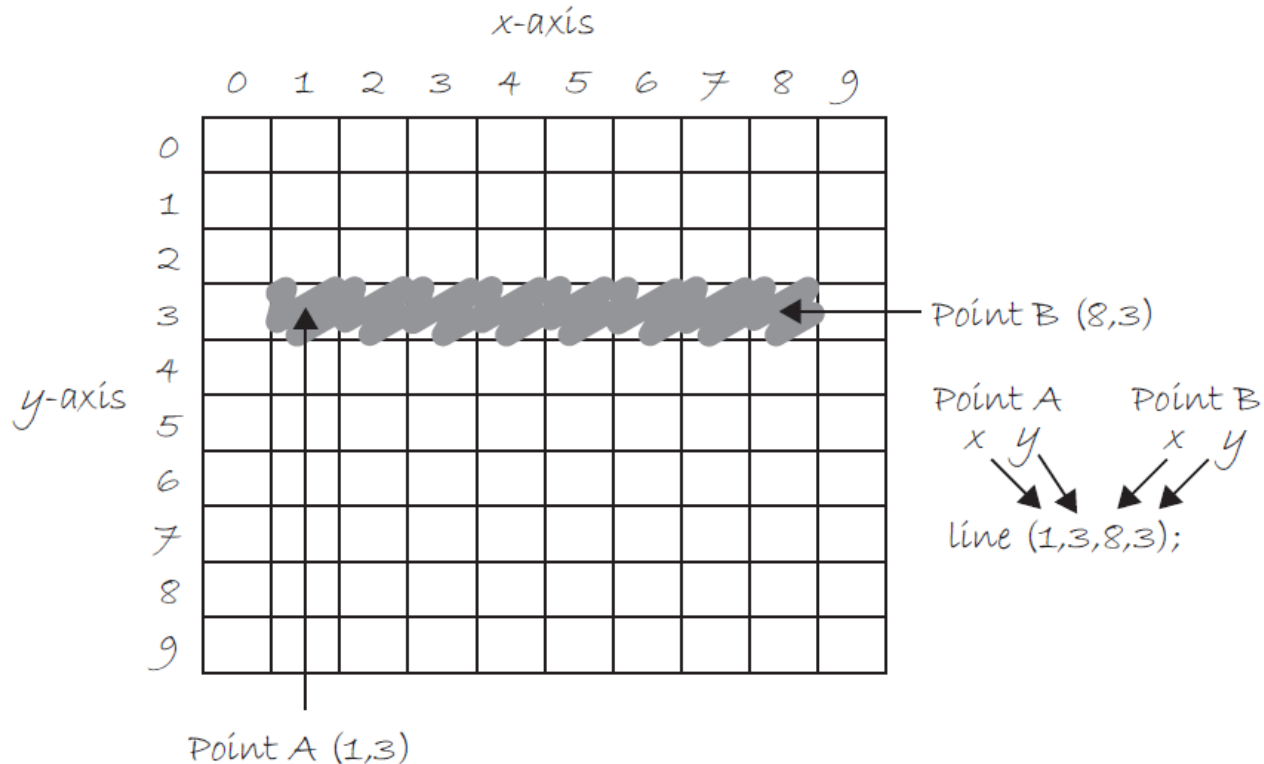
Note: We are assuming a window with a width of 10 pixels and height of 10 pixels. This is not particularly realistic since we will most likely work with much larger windows (10x10 pixels is a few millimeters of screen space).



Lines

- ▶ A line requires **two points** for being drawn.

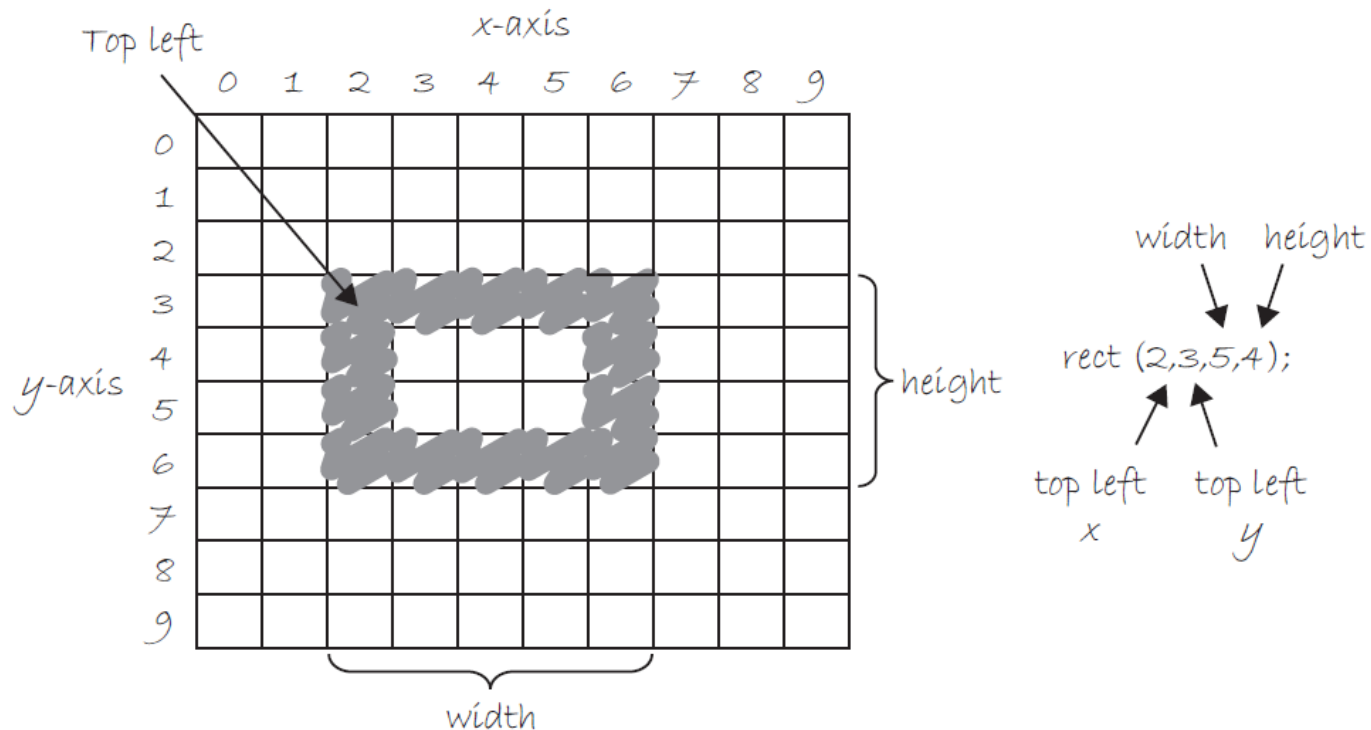
```
line (x1, y1, x2, z2) ;
```



Rectangles

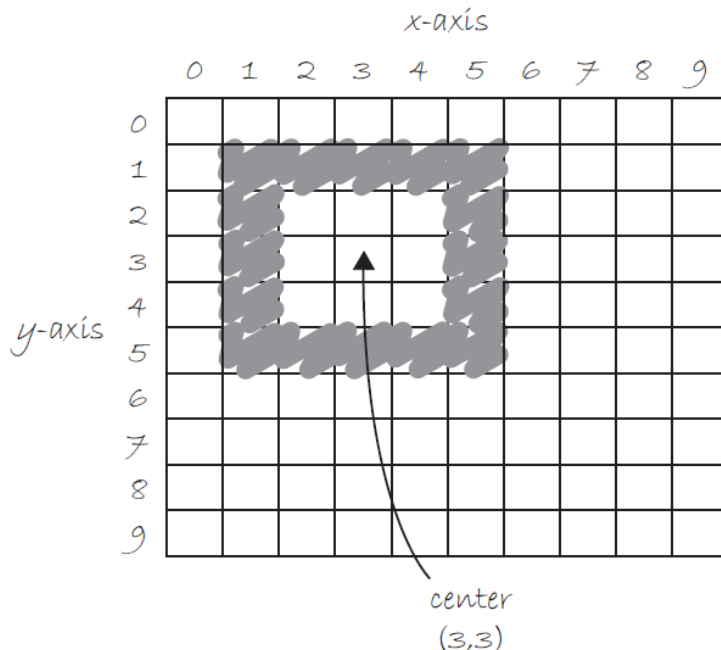
- ▶ To draw a rectangle, you must specify the coordinate for the **top left corner** of the rectangle, as well as its **width** and **height**.

```
rect (x, y, w, h) ;
```



Rectangles

- ▶ A second way to draw a rectangle involves specifying the **center point**, along with **width** and **height**.
- ▶ If we prefer this method, we first indicate that we want to use the **CENTER mode** before the instruction for the rectangle itself (the default mode is “CORNER”).



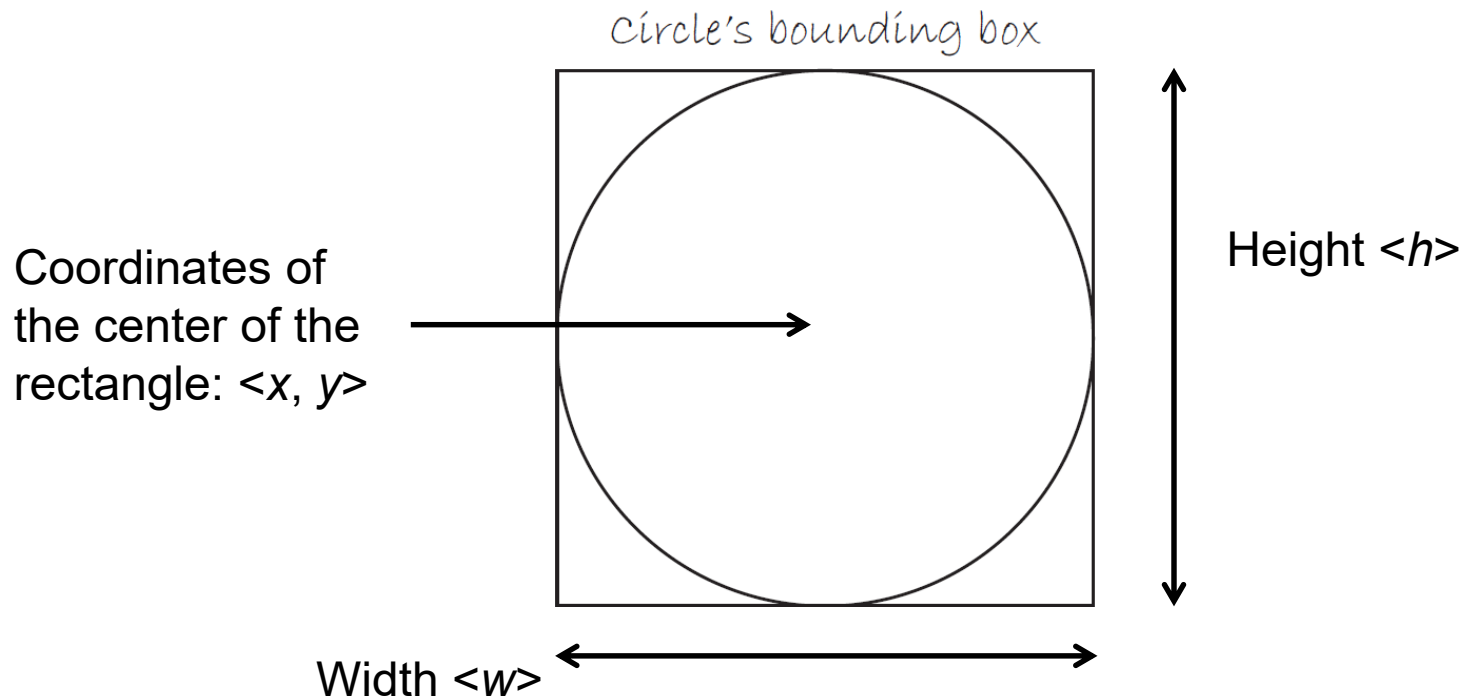
```
rectMode(CENTER);  
rect(3,3,5,5);
```

center x center y height width

```
rectMode(CENTER);  
rect(x, y, w, h);
```

Ellipses

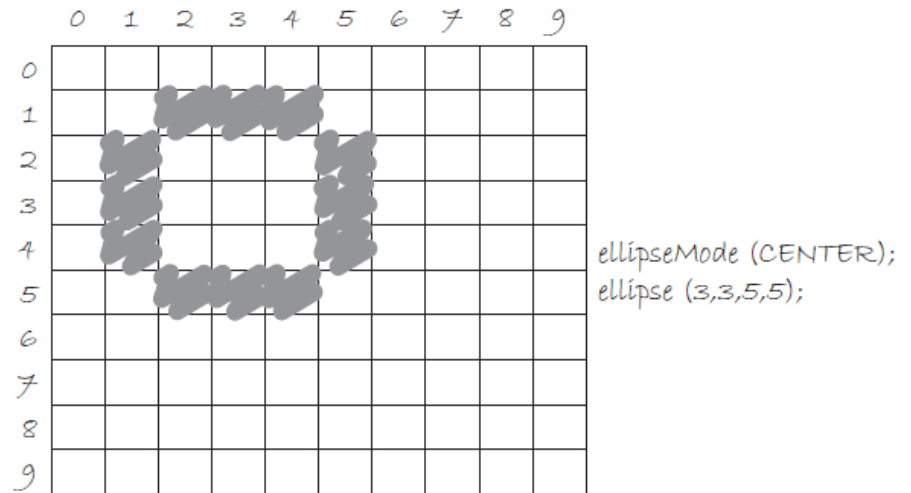
- ▶ Designing an ellipse is identical to `rect (...)`
- ▶ An ellipse is drawn where the bounding box of the rectangle would be. The default mode for `ellipse(x,y,w,h)` is “CENTER”.



Ellipses

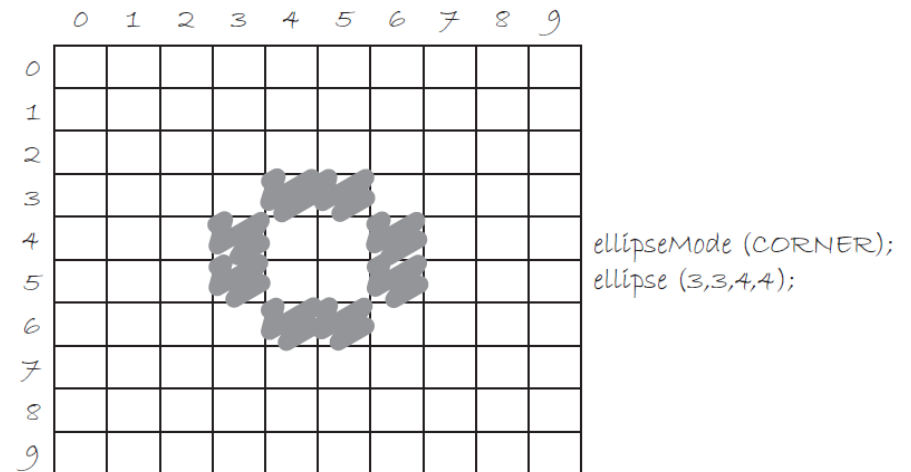
- ▶ Ellipse in mode “CENTER”

```
ellipseMode(CENTER);  
ellipse(3,3,5,5);
```



- ▶ Ellipse in mode “CORNER”

```
ellipseMode(CORNER);  
ellipse(3,3,4,4);
```



Other primitive shapes

- ▶ A **triangle** is a plane created by connecting three points.
 - ▶ The first two arguments (x1,y1) specify the first point, the middle two arguments (x2,y2) specify the second point, and the last two arguments (x3,y3) specify the third point.

```
triangle(x1, y1, x2, y2, x3, y3);
```

- ▶ A **quad** is a quadrilateral. It is similar to a rectangle, but the angles between its edges are not constrained to ninety degrees.
 - ▶ The first pair of parameters sets the first vertex and the subsequent pairs should proceed clockwise or counter-clockwise around the defined shape.

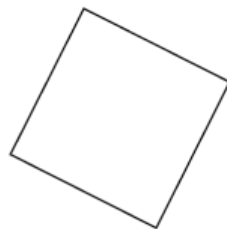
```
quad(x1, y1, x2, y2, x3, y3, x4, y4);
```



Triangle



Arc



Quad



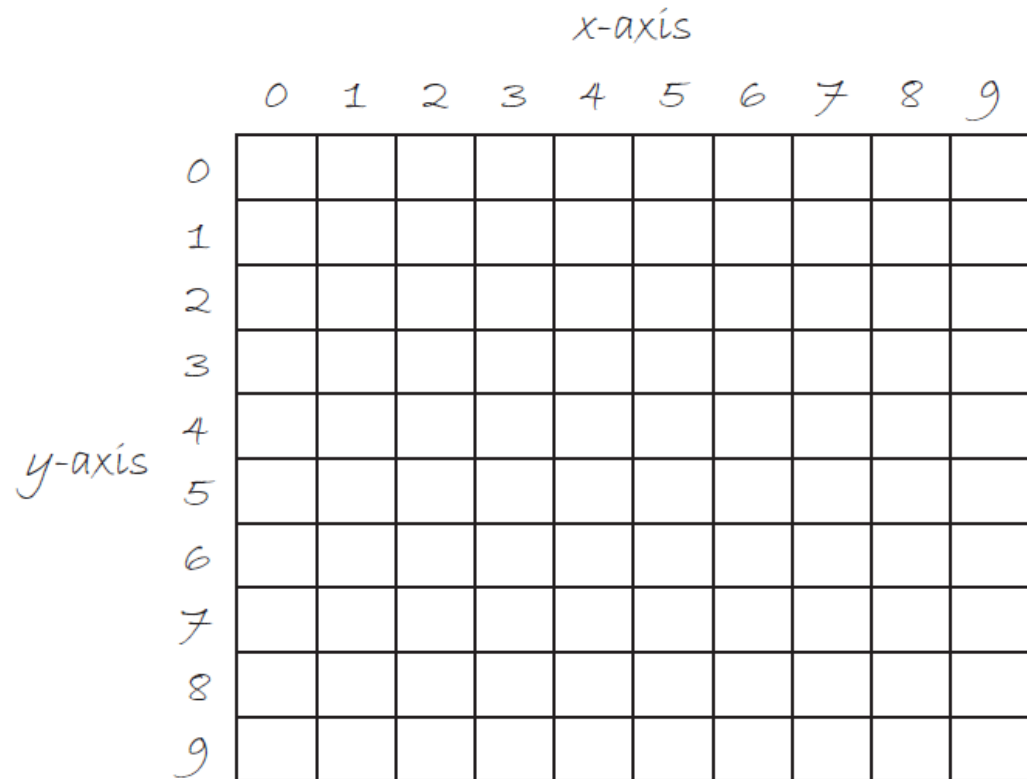
Curve

Syntax of any available shape:
<https://processing.org/reference/>

Exercise 1

- ▶ Using the blank graph below, draw the primitive shapes specified by the code.

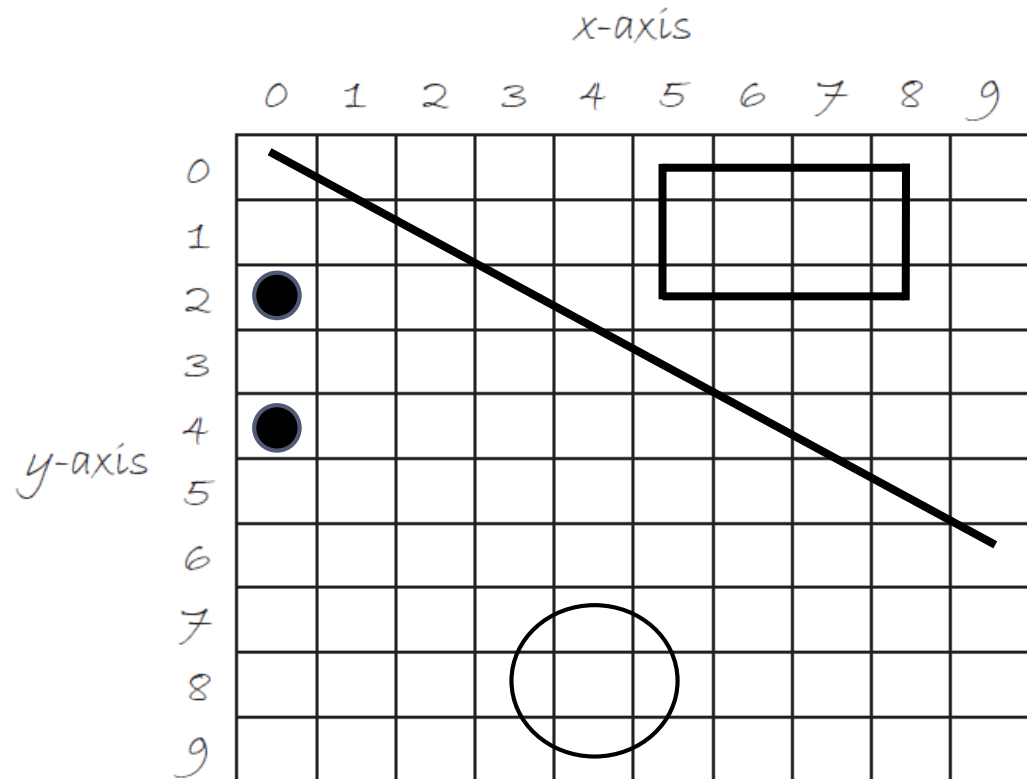
```
line(0,0,9,6);  
point(0,2);  
point(0,4);  
rectMode(CORNER);  
rect(5,0,4,3);  
ellipseMode(CORNER);  
ellipse(3,7,3,3);
```



Solution to Exercise 1

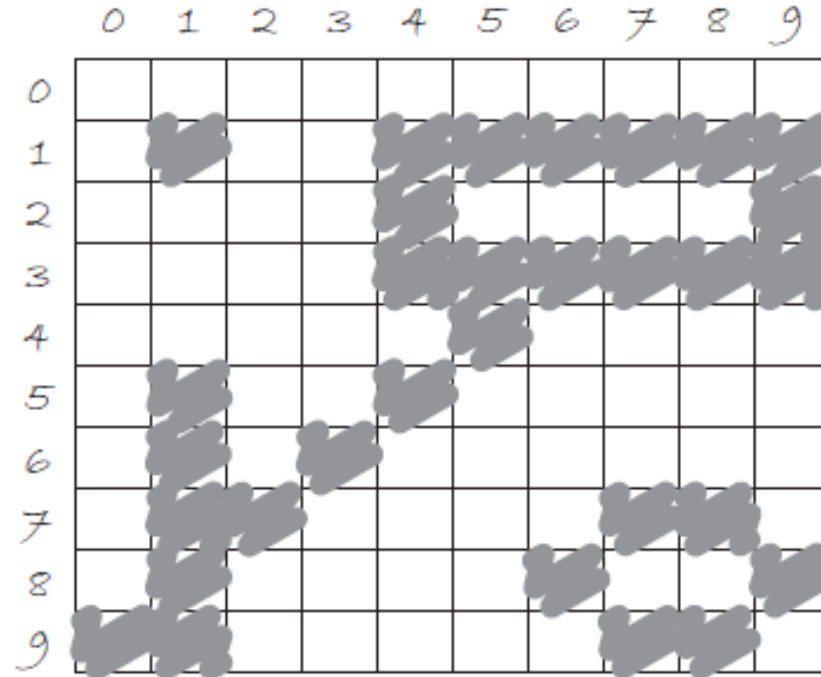
- ▶ Using the blank graph below, draw the primitive shapes specified by the code.

```
line(0,0,9,6);  
point(0,2);  
point(0,4);  
rectMode(CORNER);  
rect(5,0,4,3);  
ellipseMode(CORNER);  
ellipse(3,7,3,3);
```



Exercise 2

- ▶ Reverse engineer a list of primitive shape drawing instructions for the diagram below.
 - ▶ **Note:** There is more than one correct answer!

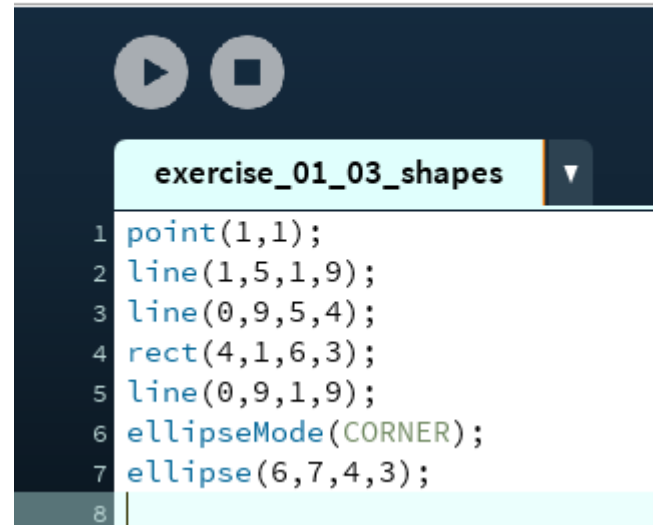


A Possible Solution to Exercise 2

```
point(1,1);  
line(1,5,1,9);  
line(0,9,5,4);  
rect(4,1,6,3);  
line(0,9,1,9);  
ellipseMode(CORNER);  
ellipse(6,7,4,3);
```

exercise_01_03_shapes | Processing 3.3

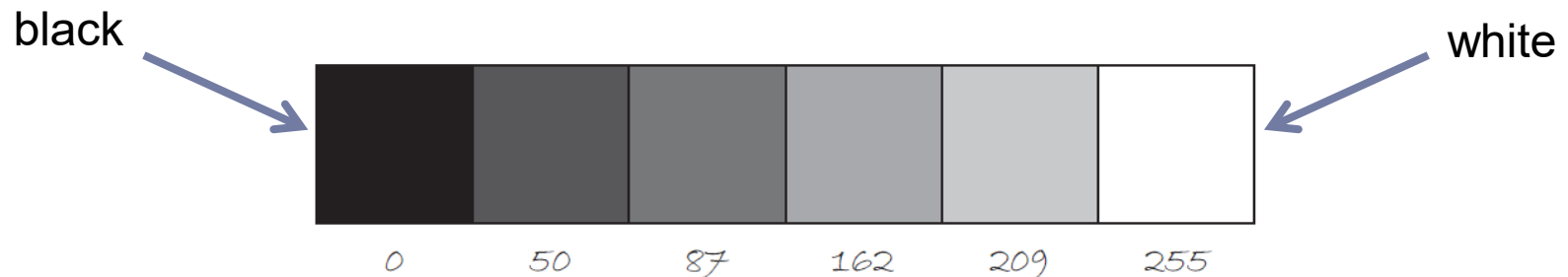
File Edit Sketch Debug Tools Help



```
exercise_01_03_shapes ▾  
1 point(1,1);  
2 line(1,5,1,9);  
3 line(0,9,5,4);  
4 rect(4,1,6,3);  
5 line(0,9,1,9);  
6 ellipseMode(CORNER);  
7 ellipse(6,7,4,3);  
8
```

Grayscale colors

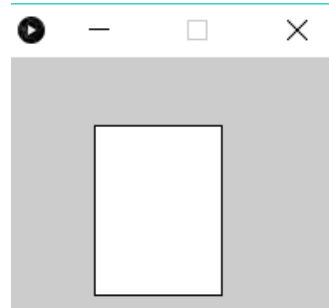
- ▶ In Processing, color is defined with a **range of numbers**.
- ▶ Simplest case: **black, white** or **grayscale**.



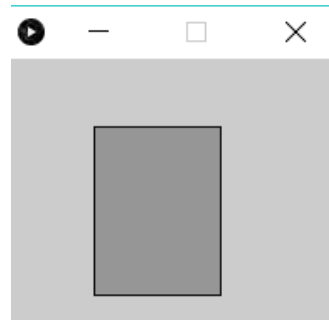
- ▶ Specific colors can be used for setting any designed shape.
- ▶ In Processing, any shape has a `stroke(n)` or `fill(n)` or both.
 - ▶ `n` is the number associated to the color.
- ▶ `stroke(n)` specifies the color for the outline of the shape
- ▶ `fill(n)` specifies the color for the interior of the shape
- ▶ Lines and points can only have `stroke(n)`.

Using grayscale colors

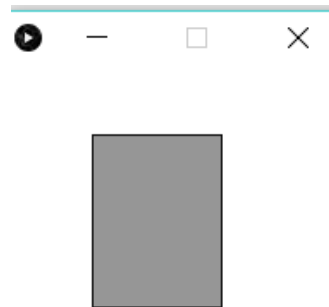
- ▶ If you forget to specify a color, Processing uses black ($n=0$) for the `stroke(n)` and white ($n=255$) for the `fill(n)` by default.
- ▶ By adding the `stroke(n)` and `fill(n)` functions before the shape is drawn, it is possible to set the color.
- ▶ There is also a function `background(n)` that sets a background color for the window (that by default is grey).



```
size(200,200);  
rect(50,40,75,100);
```



```
size(200,200);  
stroke(10);  
fill(150);  
rect(50,40,75,100);
```

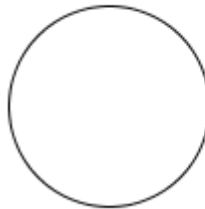


```
size(200,200);  
background(255);  
stroke(10);  
fill(150);  
rect(50,40,75,100);
```

noFill() and noStroke()

- ▶ `stroke(n)` or `fill(n)` can be eliminated with the `noStroke()` or `noFill()` functions.
- ▶ Our instinct might be to say `stroke(0)` for no outline...
- ▶ ...however, it is important to remember that 0 is not “nothing”, but rather denotes the color **black**.

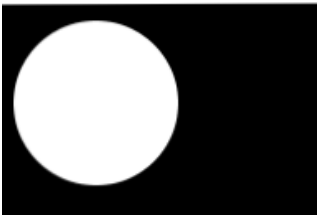
- ▶ Example of `noFill()`:



`noFill()` leaves the shape with only an outline

```
size(200,200);  
background(255);  
stroke(0);  
noFill();  
ellipse(60,60,100,100);
```

- ▶ Example of `noStroke()`:



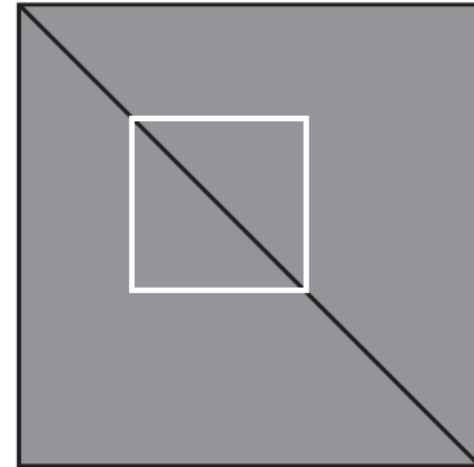
```
size(200,200);  
background(0);  
fill(255);  
noStroke();  
ellipse(60,60,100,100);
```


More shapes at one time

- ▶ If we draw two shapes at one time, Processing will always use the most recently specified `stroke(n)` and `fill(n)`, **reading the code from top to bottom**.

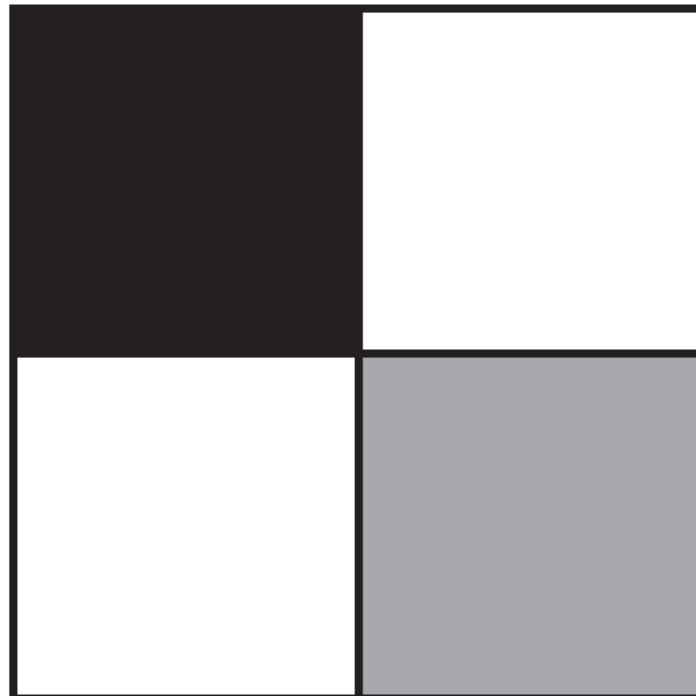
```
background(150);  
stroke(0);  
line(0,0,100,100);  
stroke(255);  
noFill();  
rect(25,25,50,50);
```

The code above is annotated with arrows pointing to the `stroke(0);`, `stroke(255);`, and `rect(25,25,50,50);` lines, indicating the order of drawing operations.



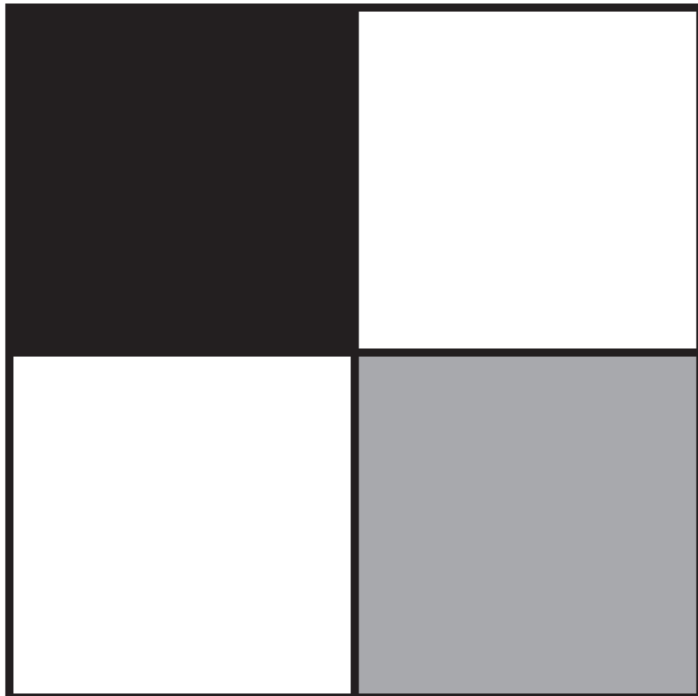
Exercise 3

- ▶ Try to guess what the instructions would be for the following screenshot.



Solution to Exercise 3

- ▶ Try to guess what the instructions would be for the following screenshot.



```
size (200, 200) ;  
fill (0) ;  
rect (0, 0, 100, 100) ;  
fill (255) ;  
rect (100, 0, 100, 100) ;  
fill (255) ;  
rect (0, 100, 100, 100) ;  
fill (150) ;  
rect (100, 100, 100, 100) ;
```

RGB colors

- ▶ Digital colors are constructed by mixing three primary colors: **red**, **green** and **blue**.
- ▶ As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order **R**, **G**, and **B**.

```
fill(R, G, B) stroke(R, G, B) background(R, G, B)
```

```
size(300, 300);  
background(255, 255, 255);  
noStroke();  
fill(255, 0, 0);  
ellipse(60, 100, 50, 50);  
fill(127, 0, 0);  
ellipse(140, 100, 50, 50);  
fill(255, 200, 200);  
ellipse(220, 100, 50, 50);
```



Color transparency

- ▶ There is an optional fourth component, referred to as the color's **alpha**.
- ▶ Alpha means **transparency** and its values range from 0 to 255, with 0 being completely transparent (i.e., 0% opaque) and 255 completely opaque (i.e., 100% opaque).

`fill(R, G, B, α)` `stroke(R, G, B, α)` `background(R, G, B, α)`

```
background(0);
```

```
noStroke();
```

```
fill(0, 0, 255);
```

```
rect(0, 0, 100, 200);
```

```
fill(255, 0, 0, 255);
```

```
rect(0, 0, 200, 40);
```

```
fill(255, 0, 0, 191);
```

```
rect(0, 50, 200, 40);
```

```
fill(255, 0, 0, 127);
```

```
rect(0, 100, 200, 40);
```

```
fill(255, 0, 0, 63);
```

```
rect(0, 150, 200, 40);
```

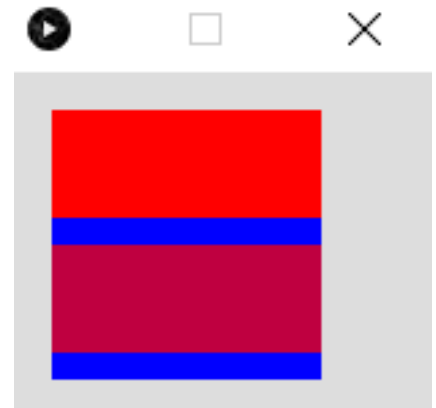
No fourth argument means 100% opacity.

255 means 100% opacity.

191 means 75% opacity.

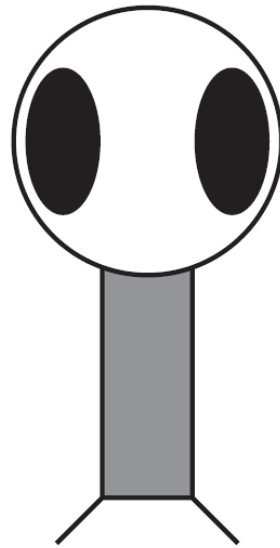
127 means 50% opacity.

63 means 25% opacity.



Exercise 4

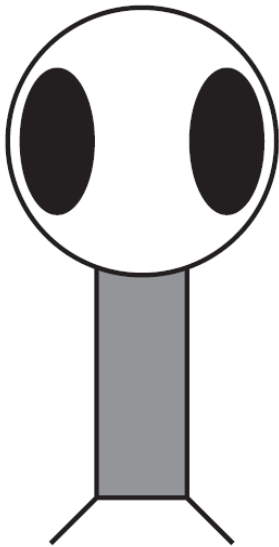
Design the following alien using simple shapes and colors:



Write the code for the alien using only points, lines, rectangles, and ellipses, using the Processing commands covered until now.

Solution to Exercise 4

Design the following creature using simple shapes and colors:



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

Go with the flow

- ▶ If you have ever played a computer game or interacted with a digital art installation, you have probably given little thought to the fact that the software that runs these experiences happens over a **period of time**. There is a **flow of events** over time.
 - ▶ The game starts with a set of initial conditions: you name your character, you start with a score of zero, and you start on level one.
 - ▶ Let's think of this part as the program's **SETUP**.
 - ▶ After these conditions are initialized, you begin to play the game.
 - ▶ At every instant, the computer checks what you are doing with the mouse, calculates all the appropriate behaviors for the game characters, and updates the screen to render all the game graphics.
 - ▶ This cycle of calculating and drawing happens over and over again, ideally **30 or more times per second for a smooth animation**.
 - ▶ Let's think of this part as the program's **DRAW**.
- ▶ The concept of flow is crucial to **move beyond static designs and sketches**.

Blocks of code

- ▶ Dynamicity is accomplished by writing two **blocks of code**: `setup()` and `draw()`.

- ▶ ***What is a block of code?***

- ▶ A block of code is any code enclosed within curly brackets.

```
{  
  A block of code  
}
```

It allows us to separate and manage our code as **individual pieces** of a larger puzzle.

- ▶ Blocks of code can be nested within each other, too.

```
{  
  A block of code  
    {  
      A block inside a block of code  
    }  
}
```

A programming convention is to **indent** the lines of code within each block to make it readable.

setup() & draw()

- ▶ **Dynamicity is accomplished by writing two blocks of code: setup() and draw().**

```
void setup() {  
  
  // Initialization code goes here  
  
}  
  
void draw() {  
  
  // Code that runs forever goes here. This block  
  // tells to the system what to draw on our screen.  
}
```

setup() & draw()

- ▶ Dynamicity is accomplished by writing two **blocks of code**: `setup()` and `draw()`.

```
void setup()
```

```
{
```

Curly brackets open and close a block of code

```
// Initialization code goes here
```

All statements preceded by `//` are “comments” that the program will ignore.

```
}
```

```
void draw()
```

```
{
```

```
// Code that runs forever goes here. This block  
// tells to the system what to draw on our screen.
```

```
}
```

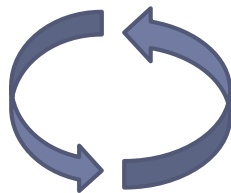
Behaviour of `setup()` & `draw()`

- ▶ **Dynamicity** is accomplished by writing two **blocks of code**: `setup()` and `draw()`.

```
void setup() {  
  // Step 1a  
  // Step 1b  
  // Step 1c  
}
```



```
void draw() {  
  // Step 2a  
  // Step 2b  
}
```



When we run the program, it will follow our instructions precisely, executing:

- the steps in `setup()` first, one time.
- and then loop continuously on to the steps in `draw()`, until the sketch is closed.

The order ends up being something like:
1a, 1b, 1c, 2a, 2b, 2a, 2b, 2a, 2b, 2a,
2b, 2a, 2b, 2a, 2b ...

The alien meets setup() & draw()

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

size(w,h) should always
be first line of setup()

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

If we run this code, **nothing in the window changes**. This looks identical to a static sketch! **What is going on?**

Nothing in the draw() function **varies**. Each time the program cycles through the code and executes the identical instructions. So, **yes**, the program is running over time redrawing the window, but it looks static to us since it draws the same thing each time!



Variation with the Mouse

- ▶ What if, instead of typing a number into one of the drawing functions, you could type “*the mouse’s X location*” or “*the mouse’s Y location*”?
- ▶ You must use the keywords `mouseX` and `mouseY`, indicating the horizontal or vertical position of the mouse cursor.

```
void setup() {  
    size(200,200);  
}  
void draw() {  
    background(255);  
    fill(175);  
    rectMode(CENTER);  
    rect(mouseX,mouseY,50,50);  
}
```

`background()` erases the window during any cycle of `draw()`.

The rectangle will be re-created each time the mouse pointer is moved.

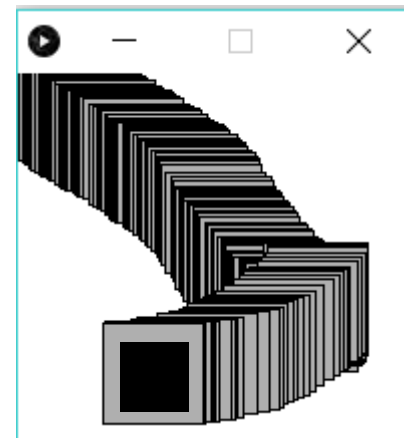


Variation with the Mouse

- ▶ Try moving `background(n)` to `setup()` and see the difference!

```
void setup() {  
    size(200,200);  
    background(255);  
}  
  
void draw() {  
    // Body  
    stroke(0);  
    fill(175);  
    rectMode(CENTER);  
    rect(mouseX,mouseY,50,50);  
}
```

`background()` is invoked just once when the sketch starts.



Note: The updating of the window happens only at the end of every cycle of `draw()`.

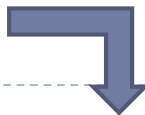
The alien starts to move

```
void setup() {  
  // Set the size of the window  
  size(200,200);  
}  
  
void draw() {  
  background(255); // Draw a white background  
  // Set ellipses and rectangles to CENTER mode  
  ellipseMode(CENTER);  
  rectMode(CENTER);  
  // Draw alien's body  
  fill(150);  
  rect(mouseX,mouseY,20,100);  
  // Draw alien's head  
  fill(255);  
  ellipse(mouseX,mouseY-30,60,60);  
}
```

Alien's body is drawn at location
(mouseX, mouseY).

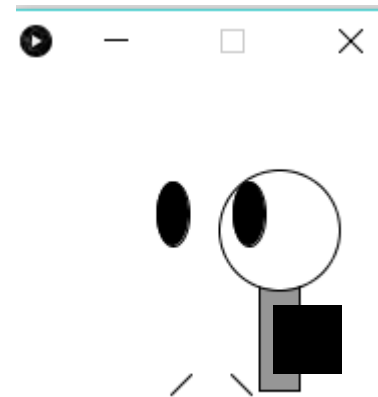
Alien's head is drawn above the body
at location (mouseX, mouseY-30).

...continue...



The alien starts to move

```
// Draw alien's eyes  
fill(0);  
ellipse(81,70,16,32);  
ellipse(119,70,16,32);  
  
// Draw alien's legs  
line(90,150,80,160);  
line(110,150,120,160);  
}
```

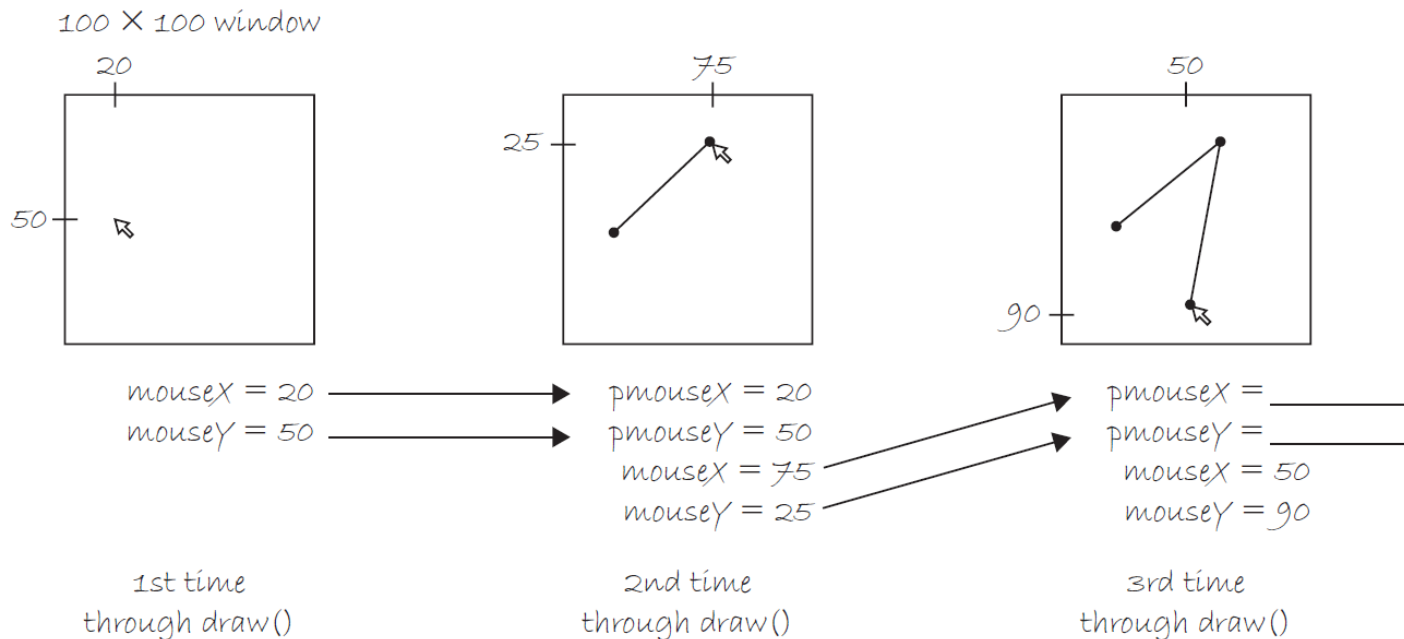


Alien's legs and eyes are maintained fixed on the screen.



Variation with the Mouse

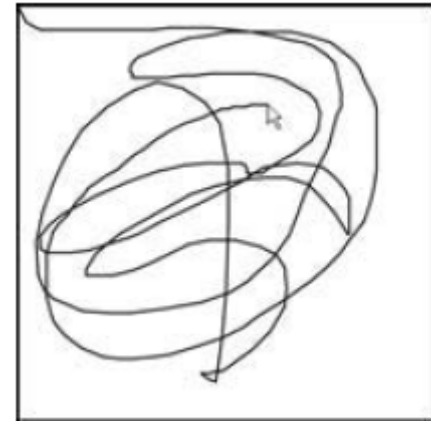
- ▶ In addition to `mouseX` and `mouseY`, we can also use `pmouseX` and `pmouseY`.
- ▶ These keywords stand for the “previous” `mouseX` and `mouseY` locations, that is, where the mouse was the last time we cycled through `draw()`.
- ▶ For example, let’s consider what happens if we draw a line from the previous mouse location to the current mouse location.



Drawing a continuous line

- ▶ By connecting the previous mouse location to the current mouse location with a line each time through `draw()`, we are able to render a **continuous line** that follows the mouse.

```
void setup() {  
  size(200,200);  
  background(255);  
}  
  
void draw() {  
  stroke(0);  
  line(pmouseX ,pmouseY ,mouseX ,mouseY);  
}
```



Mouse Clicks and Key Presses

- ▶ We know `setup()` happens once and `draw()` loops forever. **When does a mouse click occur?** Mouse presses (and key presses) are considered **events** in Processing .
- ▶ If we want **something to happen** (such as “*the background color changes to red*”) when the mouse is clicked (or a key is pressed), we need to add a third block of code to handle this event.
- ▶ Specifically, we need two new functions:
 - ▶ `mousePressed()` - Handles mouse clicks.
 - ▶ `keyPressed()` - Handles key presses.

Mouse Clicks and Key Presses

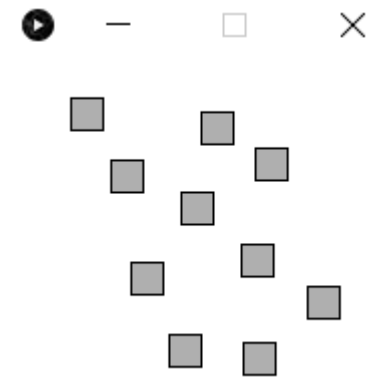
- ▶ In the following example, we have 5 functions that describe the program's flow.
 - ▶ The program starts in `setup()` where the size and background are initialized.
 - ▶ It continues into `draw()`, looping endlessly. Since it contains no code, the window remains blank. We have added two new functions: `mousePressed()` and `keyPressed()`.
 - ▶ The code inside these functions sits and waits. When the user clicks the mouse (or presses a key), it springs into action, executing the block of instructions **once** and **only once**.

```
void setup() {  
  size(200,200);  
  background(255);  
}  
  
void draw() {}  
  
void mousePressed() {  
  fill(175);  
  rectMode(CENTER);  
  rect(mouseX,mouseY,16,16);  
}  
  
void keyPressed() {  
  background(255);  
}
```

Nothing happens in `draw()` in this example!

Whenever a user clicks the mouse the code written inside `mousePressed()` is executed.

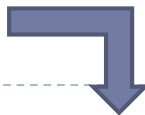
Whenever a user press a key the code written inside `keyPressed()` is executed.



The dynamic alien

```
void setup() {  
  // Set the size of the window  
  size(200,200);  
}  
  
void draw() {  
  background(255); // Draw a white background  
  // Set ellipses and rectangles to CENTER mode  
  ellipseMode(CENTER);  
  rectMode(CENTER);  
  // Draw alien's body  
  fill(150);  
  rect(mouseX,mouseY,20,100);  
  // Draw alien's head  
  fill(255);  
  ellipse(mouseX,mouseY-30,60,60);  
}
```

...continue...



The dynamic alien

```
// Draw alien's eyes
fill(mouseX,0,mouseY);
ellipse(mouseX-19,mouseY-30,16,32);
ellipse(mouseX + 19,mouseY-30,16,32);

// Draw alien's legs
line(mouseX-10,mouseY + 50,pmouseX-10,pmouseY + 60);
  line(mouseX + 10,mouseY + 50,pmouseX + 10,pmouseY + 60);
}

void mousePressed() {
  println( "Take me to your leader! ");
}
```

The eye color is determined by the mouse location.

When the mouse is clicked, a message will be displayed in the message window: "Take me to your leader!"

The legs and the eyes are drawn according to the mouse location and the previous mouse location.