02.10.18 //

PROCESSING II

DART 631
RILLA KHALED

SETUP, LOOP, AND REACT.

Processing is actually *always* keeping track of where the mouse is in its window as an (x,y) coordinate in pixels.

It stores the latest (x,y) coordinate in two variables called

`mouseX` and `mouseY`

If you use `mouseX` and `mouseY` somewhere in your program, they will be replaced by the x and y coordinates of the mouse respectively.

Let's look at some code.

```
void setup()
{
  size(500,500);
  background(255);
  rectMode(CENTER);
}

void draw()
{
  rect(mouseX,mouseY,20,20);
}
```

how do we get rid of the weird tracing?

SETUP, LOOP, AND REACT.

In fact Processing is also always keeping track of where the mouse *was* is in its window in the last frame as another (x,y) coordinate in pixels.

It stores the previous (x,y) coordinate in two variables called

`pmouseX` and `pmouseY`

If you use `pmouseX` and `pmouseY` somewhere in your program, they will be replaced by the x and y coordinates of where the mouse was in the last frame respectively.

Let's look at some code.

```
void setup()
{
  size(500,500);
  background(255);
  stroke(255,0,0);
}

void draw()
{
  line(pmouseX,pmouseY,mouseX,mouseY);
}
```

SETUP, LOOP, AND REACT.

With other kinds of input, we can't continuously track something like the location of the mouse.

Instead, sometimes we want to react only *when* something happens, like a mouse click or a key press.

To do this, Processing has special functions, just like `setup()` and `draw()`, that only run when the appropriate event has happened.

Those kinds of functions are often called "event handlers".

"Special procedures"

"Emergency protocols"

SETUP, LOOP, AND REACT.

If we want to do something when the mouse button gets pressed down, we can use this function:

```
void mousePressed(){
  // Do something!
  // The mouse button got pressed!
}
```

Here's an example – notice how you **need** to have that empty draw() loop or it won't work!

```
void draw()
{
}

void mousePressed()
{
  rect(mouseX,mouseY,20,20);
}
```

SETUP, LOOP, AND REACT.

If we want to do something when the mouse button gets let go after a click, we can use this function:

```
void mouseReleased(){
  // Do something!
  // The mouse button got released!
}
```

Same thing, but now the rectangle appears after you let go of the mouse button – these things matter.

```
void draw()
{
}

void mouseReleased()
{
  rect(mouseX,mouseY,20,20);
}
```

SETUP, LOOP, AND REACT.

If we want to do something when a keyboard key gets pressed, we can use this function:

```
void keyPressed(){
  // Do something!
  // A key got pressed!
}
```

Same thing, but now the rectangle appears if you press a key down

```
void draw()
{
}

void keyPressed()
{
  rect(mouseX,mouseY,20,20);
}
```

SETUP, LOOP, AND REACT.

If we want to do something when a keyboard key gets released, we can use this function:

```
void keyReleased(){
  // Do something!
  // A key got released!
}
```

```
Same thing, but now the rectangle appears if you let go of a key

void draw()
{
}

void keyReleased()
{
  rect(mouseX,mouseY,20,20);
}
```

SETUP, LOOP, AND REACT.

So now we can get setup, do something over and over, and do some things in reaction to the user's input!

```
void setup() {
  // Get set up
}

void draw() {
  // Do this over and over
}

void mousePressed(){
  // Do this when the mouse is pressed
}
```

This is now some quite powerful stuff – a huge amount of control over your program already.

CODING INTERACTIONS

Our programs are now reactive. But we can't always know what the user will do. We have to start planning ahead for what they might do.

In short, we're making little worlds. We set them in motion, and then other people come and interact with them.

If we want control over these worlds, we have to be able to predict what people will do, and plan for what they might want to do.

SHORT EXERCISE

1/ Play around with the `setup()` and `draw()` functions along with some of the interactive variables like `mouseX` and `mouseY` and functions like `mousePressed(){}` and `keyPressed(){}`

2/ Over this process, build up an interesting image that is interactive in some way.

Can you make more than one shape follow the mouse?

Can you make a shape go the opposite way from the mouse?

Can you make things change colours based on where the mouse is? (Check out `map()` in the Reference and see if that helps…)

VARIABLES

why variables?

analogy

NUMBERS ARE ANNOYING.

THEY'RE TOO PERMANENT.

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(175,200,25,150);
rect(300,200,25,150);
rect(200,400,25,100);
rect(275,400,25,100);
```

Which numbers are the legs? How do I change the length of the legs? How do I position the avatar somewhere else on the screen?

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(200,400,25,100);
rect(275,400,25,100);
```

THEY'RE TOO PERMANENT.

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(175,200,25,150);
rect(300,200,25,150);
rect(200,400,25,100);
rect(275,400,25,100);
```

Hardcoded numbers are a huge pain to change and are largely meaningless when you're reading your code.


```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(200,400,25,100);
rect(275,400,25,100);
```

THEY'RE TOO PERMANENT.

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(175,200,25,150);
rect(300,200,25,150);
rect(200,400,25,100);
rect(275,400,25,100);
```

What if I wanted to gradually move the avatar around on the screen from frame to frame (that is, change its location in a `draw()` loop?)


```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(200,400,25,100);
rect(275,400,25,100);
```

THEY'RE TOO PERMANENT.

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(175,200,25,150);
rect(300,200,25,150);
rect(200,400,25,100);
rect(275,400,25,100);
```

Hardcoded numbers cannot be changed while the program is running, they're completely fixed.


```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(200,400,25,100);
rect(275,400,25,100);
```

A VARIABLE IS LIKE AN X.

A variable is like a place to store something you want to remember, refer back to, and change.

It's like a box, it's like a pigeon hole, it's like a napkin, it's like a sticky, it's like a bucket, it's like a drawer, it's like a thingamajig, it's like a … name for a place you store a value.

Three key qualities:

A name, a type, and a value.

A NAME.

Variables have names.

Names have meanings.

`mouseX` means “the pixel on the x-axis where the mouse is pointing right now”.

Give meaningful names.

mouseX is a name given by the Processing language - a variable with a name that we know we can access the value of.

You can create your own variables of course, but choose descriptive names that will help you remember what the variable represents.

A TYPE.

In (most) programming languages, you have to say what kind or *type* of value a variable is made to hold.

Like an integer, a color, or a string of letters, say.

By saying what *type* a variable has, you're letting the programming language know what kinds of things you will do with it.

You don't multiply a letter by a number, but you do multiply two numbers, for instance.

There is such a thing as an "untyped" language, where you don't have to care about what type a variable is.

But you do have to care in Processing. So start caring.

Now we're in the **ontology of programming** – the idea of a hierarchy or taxonomy of "things".

A VALUE.

That is, the thing you're actually storing in the variable. The actual number, character, color, etc.

Importantly, the variable will *keep* that value safe for you until you need it again.

Even more importantly, you can *change* the value inside a variable while the program is running.

```
int meaningOfLife = 42;
```

This is a *variable declaration*.

This is how you tell Processing you want a variable that you can store something in (in this case it's an integer variable).

```
int meaningOfLife = 42;
```

First we write the type of the variable.

In this case we have `int`, which stands for “integer”. That is, whole numbers that can be either positive or negative.

Other key types include:

`float` – a “floating point” number, like 3.14159

`char` – a single character, like ‘a’, ‘b’, or ‘%’

`String` – a string of characters, like “Hello, World!”

`color` – a color!

```
int meaningOfLife = 42;
```

Next we write the name of the variable.

In this case we are calling it `meaningOfLife`.

Notice how this name explains what the variable will store, so you don't have to wonder.

Also notice the convention of starting with a lowercase letter and then using uppercase to show the start of new words (known as camelCase).

```
int meaningOfLife = 42;
```

Next we write the assignment operator which uses the symbol =.

This says to Processing: “Now I am about to tell you what to put inside my new integer variable called `meaningOfLife`.” I think of ‘=‘ like ‘is’, i.e. `meaningOfLife` is _____.

Notice how confusing it is to use an equals sign here, and prepare yourself to find it even more confusing later, when you learn that the symbol for equals is ==.

```
int meaningOfLife = 42;
```

Next we tell Processing what value we want to store in the integer variable called `meaningOfLife`.

Note that we have to specify the *right type* of value here. We can't ask it to store a value like `"forty two"`, because that isn't an integer, that's a string of characters.

```
int meaningOfLife = 42;
```

Here are some appropriate values for the other types:

```
float theNumberPi = 3.14159;
char theCharacterA = ‘a’;
String helloWorld = “Hello, World!”;
color red = color(255,0,0);
```

Notice how we use special characters to show what type of value we are using. A `float` uses a period, a **char** has single quote marks around the character, a `String` has double quote marks around the string, and a color uses that weird `color(r,g,b)` syntax;

```
int meaningOfLife;
```

This is another way of *declaring a variable*.

In this case, you are telling Processing to make the integer variable `meaningOfLife` exist, but without telling it what value it should start off with.

Because maybe your program will work out the meaning of life later on.

`int meaningOfLife;`

But then, what is inside **`meaningOfLife`**? What would happen if you used it before it had a value in it?

Let's see…

In general, it's a good idea to *initialise* your variables when you declare them so you maintain control over what's going on.

```
There are two different things that can happen!

In one version, Processing will help out:

—

int meaningOfLife;
println(meaningOfLife);

——

In another version it will just assume the value is 0:

——

int meaningOfLife;

void setup()
{
  println(meaningOfLife);
}
```

USING VARIABLES.

Now that you have a variable, you can use it as if it were the actual thing stored inside it.

So you can use an integer variable anywhere you would use an integer, you can use a String variable anywhere you would use a String.

```
int meaningOfLife = 42;

rect(0,0,meaningOfLife,meaningOfLife);


String helloWorld = “Hello, World!”;

println(helloWorld);
```

Notice how stupid it is to draw a rectangle with the height and width of the meaning of life from a semantic perspective…

… but Processing doesn’t care! It just uses the value inside the variable, it doesn’t really mind about the name.

—

```
int meaningOfLife = 42;
rect(0,0,meaningOfLife,meaningOfLife);

String helloWorld = “Hello, World!”;
println(helloWorld);
```

ON ARITHMETIC.

Very quickly, notice that you can do arithmetic with the types of variables that are numbers (or with hardcoded numbers, of course):

```
int meaningOfLife = 21 + 21; // add

fill(meaningOfLife * 5, 0, 0); // multiply

rect(meaningOfLife/2, 0, 50, 50); // divide
```

There are more of these kinds of operators – check out the Reference in the section called “operators”.

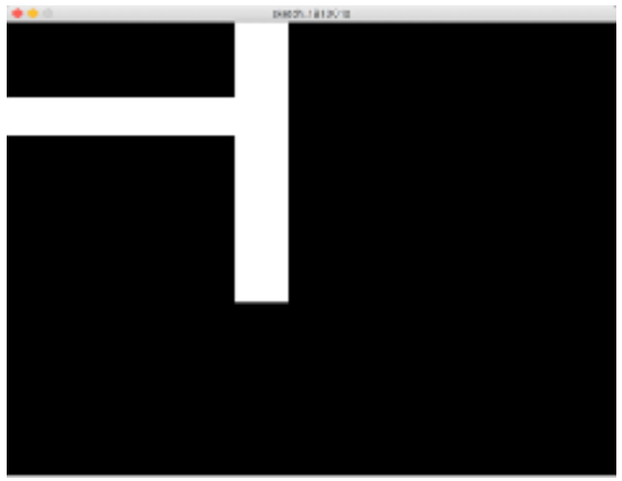
So what will this do?

```
int meaningOfLife = 21 + 21; // add
fill(meaningOfLife * 5,0,0); // multiply
rect(meaningOfLife/2,0,50,50); // divide
```

BASIC MOTION.

Frame by frame, we can be changing the value of variables, which gives us very basic animation.

Note that we are declaring our variables before setup()



```
int rect1Width = 0;
int rect2Height = 0;

void setup() {
  size(800, 600);
  background(0, 0, 0);
  println("window width: "+width +" window height: "+ height);
}

void draw() {
  // set my colors
  fill(255, 255, 255);
  noStroke();

  // draw my rectangles
  rect(0, 100, rect1Width, 50);
  rect(300, 0, 70, rect2Height);

  // change my variable values
  rect1Width = rect1Width + 1;
  rect2Height = rect2Height + 1;

  // print them out so I can see them
  println("rect1Width: "+rect1Width+" rect2Height: "+rect2Height);
}
```

BUILT-IN VARIABLES.

Your friends `mouseX` and `mouseY` are an example of built-in variables – useful variables that store useful information you might want to use!

Here are some more (look them up in the Reference!):

`width` and `height`

`frameCount` and `frameRate`

`key`, `keyCode`, and `keyPressed`

`mousePressed` and `mouseButton`

width and height = the width and height of your program's window
frameCount = the current frame "playing"
frameRate = the current frame rate of the program
key = value of most recent key pressed.
keyCode = code of most recent key pressed
keyPressed = true or false based on whether a key is being pressed mousePressed = true or false based on whether a mouse button is being pressed
mouseButton = the number of the button being pressed on the mouse

You can also look these up in the Processing Reference!

REMEMBER ME?

```
size(500,500);
rect(200,200,100,200);
rect(225,150,50,50);
rect(175,200,25,150);
rect(300,200,25,150);
rect(200,400,25,100);
rect(275,400,25,100);
```

```
int avatarX = 50;
int avatarY = 50;
int avatarHeadSize = 50;
int avatarBodyWidth = 100;
int avatarBodyHeight = 200;
int avatarLegWidth = 25;
int avatarLegHeight = 150;

void setup()
{
  size(600, 600);
}

void draw()
{
  rect(avatarX, avatarY, avatarBodyWidth, avatarBodyHeight);
  rect(avatarX + (avatarBodyWidth - avatarHeadSize)/2, avatarY - avatarHeadSize, avatarHeadSize, avatarHeadSize);
  rect(avatarX, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
  rect(avatarX + avatarBodyWidth - avatarLegWidth, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
}
```

IF WE ADD THIS TO THE DRAW LOOP?

```
avatarX = avatarX + 1;
```

Or if you want to be fancy…

```
avatarX += 1;
```

Or if you want to be really fancy…

```
avatarX++;
```

IT'S ALIIIIIIVE!

We now have created a conscious, self-sustaining life form in our program!

(Well, maybe not that – but we have managed to get what is essentially an avatar to move around.)

Pro tips:

- to get your code auto format nicely, hit "cmd + A" (select all) then "cmd T".
- COMMENTING YOUR CODE IS GOOD!

And why does it leave that trail? Oh yeah, because you need to reset the background each time!

RANDOM!

Random numbers are just so much fun in programming, and Processing gives us access to random numbers very easily with the `random()` function.

```
float randomNumber = random();
```

This will a random floating point number from 0 up to (but not including) 1 into our variable `randomNumber`.

RANDOM!

You can also specify the range you want, like this:

```
float r = random(0,255);
float g = random(0,255);
float b = random(0,255);
background(r,g,b);
```

Or even like this:

```
background(random(0,255));
```

```
void setup()
{
  size(500,500);
}

void draw()
{
  float r = random(0, 255);
  float g = random(0, 255);
  float b = random(0, 255);
  background(r, g, b);
}

====== into the same code as avatar

int avatarX = 50;
int avatarY = 50;
int avatarHeadSize = 50;
int avatarBodyWidth = 100;
int avatarBodyHeight = 200;
int avatarLegWidth = 25;
int avatarLegHeight = 150;

void setup()
{
  size(600, 600);
}

void draw()
{
  float r = random(0, 255);
  float g = random(0, 255);
  float b = random(0, 255);
  background(r, g, b);
  avatarX++;
  rect(avatarX, avatarY, avatarBodyWidth, avatarBodyHeight);
  rect(avatarX + (avatarBodyWidth - avatarHeadSize)/2, avatarY - avatarHeadSize, avatarHeadSize, avatarHeadSize);
  rect(avatarX, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
  rect(avatarX + avatarBodyWidth - avatarLegWidth, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
}
```

RANDOM!

What if we put this in the draw loop of our avatar code?

```
avatarX = (int) random(0,width);
avatarY = (int) random(0,height);
```

What the heck does `(int)` mean, you ask? It's complicated, frankly, but it means "turn the `float` that `random()` gives me into an `int`".

It's called "casting" and for now you can generally avoid it except in specific cases like this one.

```
int avatarX = 50;
int avatarY = 50;
int avatarHeadSize = 50;
int avatarBodyWidth = 100;
int avatarBodyHeight = 200;
int avatarLegWidth = 25;
int avatarLegHeight = 150;

void setup()
{
  size(600, 600);
}

void draw()
{
  background(255);
  rect(avatarX, avatarY, avatarBodyWidth, avatarBodyHeight);
  rect(avatarX + (avatarBodyWidth - avatarHeadSize)/2, avatarY - avatarHeadSize, avatarHeadSize, avatarHeadSize);
  rect(avatarX, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
  rect(avatarX + avatarBodyWidth - avatarLegWidth, avatarY + avatarBodyHeight, avatarLegWidth, avatarLegHeight);
  avatarX = (int) random(0,width);
  avatarY = (int) random(0,height);
}
```

SHORT EXERCISE

Grab the code you made last week, or start something new, and use variables to make interesting things happen:

- Try out `random()`
- Try out changing the variables during the `draw()` loop
- Try out changing the variables when the user presses a key or clicks a key or moves the mouse!

CONDITIONALS

Now we turn to the idea of controlling the flow of a program

Which is getting pretty sophisticated, and pretty awesome.

TRUE OR FALSE

In Processing, there are certain expressions that *evaluate* to be true or false, mostly based on math:

`10 < 20` is `true`

`1 + 1 == 3` is `false`

We can check on the truth or falsity of expressions like this in order to choose what to do in our programs.

It works even better if we use variables…

`avatarX < width` is … well, we don't know right now!

CONDITIONAL OPERATORS

Here are the major conditional operators:

`1 < 2` (less than)

`2 > 1` (greater than)

`1 <= 2` (less than or equal to)

`2 >= 2` (greater than or equal to)

`1 != 2` (does not equal)

`1 == 1` (equals)

All of those are… **`true`**.

THINGS ARE GETTING A LITTLE IFFY...

So how do we actually check these conditions in some useful way in our programs?

We use `if` statements.

An `if` statement checks on whether a condition is true or not, and then based on that will either do something or not.

THINGS ARE GETTING A LITTLE IFFY...

```
if ( condition )
{
     // Do something
}
```

THINGS ARE GETTING A LITTLE ELSEY...

```
if ( condition )
{
        // Do something (condition is true)
}
else
{
        // Do something else (condition is false)
}
```

We can use this extra “else” and curly brackets if we want to tell Processing what to do when the condition turned out to be `false`.

THINGS ARE GETTING A LITTLE ELSEIFFY...

```
if ( condition )
{
        // Do something (condition is true)
}
else if ( anotherCondition )
{
        // Do something (condition is false)
        // (But anotherCondition is true)
}
```

We can even specify another condition to check after we found out that `condition` is `false`.

THINGS ARE GETTING A LITTLE NESTY...

```
if ( condition )
{
        if ( anotherCondition )
        {
                // Do something
                // (condition and anotherCondition are both true)


        }
}
```

We can even *nest* conditionals to check if multiple things are true at the same time!

Because sometimes you need to know more than one thing

THINGS ARE GETTING A LITTLE LOGICAL...

We can also use logic to make more complicated conditionals, i.e. we can combine conditions. The logical operators are:

`&&` - and

`||` - or

`!` – not

So how does that work?

THINGS ARE GETTING A LITTLE LOGICAL...

`condition1 && condition2`

Is `true` if both `condition1` and `condition2` are `true`.

`condition1 || condition2`

Is `true` if *either* `condition1` or `condition2` are `true`.

`!condition1`

Is only `true` if `condition1` is `false`.

THIS SORT OF THING MIGHT BE USEFUL...

```
if (avatarX > width || avatarX < 0)
{
    // The avatar is off the screen!
}
```

So we can check important conditions in our program and then react appropriately.

```
if (avatarX > width || avatarX < 0)
{
        // The avatar is off the screen!
}

Notice that this is the same as:

if (avatarX > width)
{
        // The avatar is off the screen!
}
if (avatarX < 0)
{
        // The avatar is off the screen!
}

But better, because we only have to react one time!
```

WHAT WILL THIS DO?

```
if (2 > 0 || 10 < 9)
{
  println(“Squeezing blood from a stone!”);
}
else if (10 < 20 && 9 <= 9)
{
  println(“When pigs fly!”);
}
if (!(10 > 0 && 9 < 10))
{
  println(“When hell freezes over!”);
}
```

```
if (2 > 0 || 10 < 9)
{
  println("Squeezing blood from a stone!");
}
else if (10 < 20 && 9 <= 9)
{
  println("When pigs fly!");
}
if (!(10 > 0 && 9 < 10))
{
  println("When hell freezes over!");
}
```

BUT, AGAIN, IT'S BETTER WITH VARIABLES...

Here's something a bit more complicated…

In fact, it's the beginning of some not amazing physics…

```
int avatarX = 0; // Avatar location on X
int avatarY = 0; // Avatar location on Y
int avatarSize = 10; // Avatar size (will be a square)
int avatarVelocityX = 5; // Number of pixels avatar should move each frame on X

void setup()
{
  size(500,500);
}

void draw()
{
  background(255); // Fill the background to create animation
  avatarX += avatarVelocityX; // Add the velocity to the avatar's location so it moves
  rect(avatarX,avatarY,avatarSize,avatarSize); // Draw the avatar in its new location

  // Now check if the avatar is going off the screen
  if (avatarX > width || avatarX < 0)
  {
    // If it is, then reverse its velocity!
    avatarVelocityX = -avatarVelocityX;
  }
}
```

AND IN FACT, IT CAN BE EVEN BETTER…

Here's something even more complicated…

In fact, it's the beginning of some crappy controls…

Check the code/L5/AvatarMovement folder on GitHub for code.

MORE COMPLEX

Let's have the walls constrain the avatar.

Check the code/L5/AvatarMovementMoreComplex folder on GitHub for code.

ONE LAST SAVING GRACE!

Processing has a variable *type* which you can use to store the results of conditions in, it's called `boolean`.

The value of a `boolean` variable will be either `true` or `false`.

```
int meaningOfLife = 42;
boolean lifeHasMeaning = (meaningOfLife == 42);
if (lifeHasMeaning)
{
      // Do something!
}
```

Notice how we can use a boolean variable in the same places we would use a conditional!


```
boolean myButtonPressed = false;

void setup()
{
  size(500,500);
}

void draw()
{
  if (myButtonPressed)
  {
    background(255);
  }
  else
  {
    background(0);
  }
}

void mouseReleased()
{
  myButtonPressed = !myButtonPressed;
}
```

EXERCISE.

Modify or create a new scene to use conditionals in reacting to player input or anything else.

What would happen if you used an if statement with `random()`? (Hint: think about probability…)

What if you checked the values of `mouseX` and `mouseY` when the player clicked the mouse button?

What if you checked combinations of conditions, like where the mouse is and what key is being pressed?

TEXTBOOK ON PROCESSING

“Learning Processing” by Daniel Shiffman.

Available as an ebook from the Concordia Library.