First Sketch

Video Transcript

JON McCORMACK:

Now that you've run your first Processing sketch and drawn your name, it's time to take a look at the code that is used to create the w1_01 sketch.

We'll take a look at different sections of the code and I'll explain how they work and what they do.

Let's start at the beginning and look at something that is really important, yet its not actually used by the computer when it runs the program!

I'm talking about comments.

Commenting your code is an essential part of 'programming beautifully', even if you're the only person who will ever see that code. Comments can tell you important information about the program: what the code does and how it does it. It's easy to forget how a program works if you haven't looked at it for a while and if you share your code with others it will help them understand what your code is doing.

Its a good idea to always begin any sketch with comments that list:

- the name of the program and project it belongs to
- the author or authors
- the date the program was written
- and any copyright or license information
- a brief summary of what the program does and how to use it

There are two ways to make comments in your code. One is by beginning each comment with a / followed by an * and finishing the comment with an * followed by a /. Its important to include both these beginning and ending comment markers otherwise your program will report an error when you try to run it.

Using the /* and */ markers allows you to have comments that span multiple lines.

The second type of comment just begins with two forward-slashes, which is a little easier to type. These comments don't need any end-marker or delimiter because they always run to the end of the line on which they started. So this type of comment is useful for short, 1-line descriptions.

Now lets look at the next section of code. It contains a function called ‘setup’. Every Processing sketch your write must contain this function. It’s purpose is to tell Processing how to set everything up correctly. In computer parlance we call this ‘initialisation’.

The ‘draw your name’ sketch calls three built-in processing functions: size, background and rectMode. Notice how each function ends with a semicolon. This signifies the end of a programming statement, a bit like in written language where we end a sentence with a full stop.

size() sets the dimensions of the sketch’s display window. The two numbers inside the round brackets are called arguments. The first tells Processing the width of the window, the second the height. The sizes are in pixels. Try changing these numbers, running the sketch and see what happens.

The next function sets the background colour of the window to white. We’ll look more closely at how you specify colours in a moment.

The final function tells Processing how it should interpret function calls that draw rectangles. The capitalised RADIUS is a special Processing built-in constant. RADIUS means that you’ll specify a centre point and a radius width and height when drawing rectangles. We’ll cover drawing in more detail later.

The next section of code contains the draw function. As the name suggests, this function is used to do the actual drawing. It’s called after setup and it repeats in a loop.

I won’t go through every statement in the code, but notice the names highlighted in red. These are special Processing variables that allow the computer to tell you things about the mouse. mousePressed for example can only have 2 possible values: TRUE or FALSE. We call this a boolean variable. When the left mouse button is pressed it will have the value TRUE, if not pressed, FALSE.

mouseX and mouseY give the current position of the mouse inside the window. As you move the mouse around the values will change automatically. This is how the program is able to draw according to the movements you make with the mouse.

Every processing sketch must have these setup() and draw() functions.

setup() is run once when the program starts and is used to set the display window size, initialise variables, set drawing styles and so on.

draw() is where you do your drawing or any other processing that needs to be done repeatedly. This makes things like animation and interaction possible.

So remember: setup() is executed once at the beginning of the program’s run. draw() is called repeatedly following setup() until the program quits. To quit a running sketch press the ‘esc’ key.

Let’s look more closely at functions. Essentially, functions are blocks of code that perform a task for you.

Processing has many useful built-in functions that you can use to help you get things done. Functions are one of the fundamental building blocks of programming. You've already seen a number of built-in functions in this first sketch, including size() which sets the size of the window, background() which sets the background colour, and random() which returns a random number between 0 up to the number specified as an argument to the function.

While built-in functions are great, you can also create your own functions. This is really useful when you need to do tasks repeatedly or you need to break a complex problem down into a more manageable set of tasks.

You can call functions or define them, so that you can call them elsewhere in your code.
Let's look at calling first.

To call a function you need to type the name of the function, followed by the round brackets. Functions can take zero or more arguments, which are simply numbers or other types of information that you can send to the function for it to use. Some functions don't need any arguments, in which case you just have the pair of round brackets with nothing inside.

To define a function you need to specify a return type (in this case an integer), the name of the function and a list of parameters and their types that the function takes. This example is a function that adds ten to the argument passed. Notice that parameters can be used within the function body, and that they keyword 'return' is used to send the computed value back from the function.

Here's a list of the functions used in the 'draw your name' sketch. See if you can work out what each one does and experiment with how they work by changing the values sent to them.

To get help on any built-in Processing function or constant, select the item and right-or-control click to bring up the menu. Select 'Find in Reference' and Processing will automatically open the documentation for that function in your web browser.

Next, let's take a quick look at how colour works in Processing.

The fill function is used to set the colour that will be used to fill graphics shapes such as the rectangle that is used in the first sketch. In this example it takes a number, from 0 to 255 that specifies a grey value, with 0 corresponding to black and 255 to white. 128, which is half-way between corresponds to a mid-grey.

The second example adds an additional value, alpha, which specifies the opacity of the colour. 0 corresponds to fully transparent, 255 fully opaque.

Colour can also be specified using a mixture of red, green and blue. To help you choose a colour Processing provides a colour selector which you can access from the tools menu. Choose the colour and then use the RGB values in your code. In the example on the right I've chosen a colour in the selector and then used it to specify the fill colour and then drawn a rectangle.

In the final example, an alpha value can be added. As with the grey example this controls the opacity of the colour. The second fill call makes the next rectangle draw with a semi-transparent purple. You can see the result below the code.

It's worth pointing out that functions that control drawing colours and style remain set until you change them. Processing keeps track of a current drawing state, which includes the fill colour, the border colour, border thickness, and so on. When you call a function like rect() it uses whatever values are currently set. This means you only need to call fill when you want to change the fill colour, not every time you want to draw a filled shape.

Finally, lets take a quick look at how to deal with errors in your code. It's a normal part of programming that code you write will have errors. We'll look at two different kinds of program error: syntactic and semantic.

Syntax errors are like typographic or spelling mistakes in human languages. If your sketch has syntax errors it won't run.

Here you can see that Processing has reported an error down the bottom of the screen in red, and it has a more intelligible version in the bar above. It also highlights where it thinks the error is located in the code.

Of course this is a computer so its not always correct. It turns out the problem was a missing semicolon but the highlighted text is on the next line where processing got to when it realised there was an error.

If you encounter an error its important to remember that it may not necessarily be where Processing thinks it is. Its a good idea to keep testing a program after you make each change to the code, so when you encounter an error you'll know where to look.

The second type of error is called a semantic error. This is a gap between what the programmer wants the code to do, and what it actually does. Semantic errors are not reported when you try to run the sketch because the computer has no way of knowing what you really mean, as opposed to what you've actually written!

Sometimes a semantic error will cause the program to stop unexpectedly, or 'crash'. In other cases it may run, but no do what you want it to do.

Fixing semantic errors is much more difficult and it takes a lot of experience to find them - this is known as 'debugging' and it requires you to 'think like a computer'...