

# Looping and Repeating

## Video Transcript

### JON McCORMACK:

One thing that computers excel at doing is mindless, repetitive tasks over and over again. In fact that's one of the reasons, along with being good calculating machines, that they're so useful.

Repetition is a commonly used motif in the art world too.

In this work by the Hungarian artist Vera Molnar, the simple element of a square is repeated to great emotional effect. After working for many years in concrete and constructivist art, Molnar wanted to introduce randomness into her art. As she discovered, humans tend not to be very good at creating truly random patterns, so Molnar needed a computer to help her achieve her artistic aims. Initially, she didn't have access to one, so she built an imaginary machine in her head and mentally executed an imaginary program. Later she learned programming and could use a real computer.

We'll talk more about randomness later in the course, but for now let's deconstruct this artwork in a series of processing sketches. Along the way we're going to learn about programming constructs for looping and repeating.

Let's start by taking a look at the w2\_01, sketch. Open the sketch in processing, press "play" and look at the results. This sketch just draws nine squares, with each row coloured red, green and blue.


Taking a look at the code, we can see that the draw() function just calls "rect" 9 times – once for each square. The arguments to each rect call are different because each square is in a different location.

To create this code, I had to work out the location of each square within the display window by hand. This took a while. If I wanted to change the size of the display window, or have a different number of squares, I'd have to re-calculate the position of each one – which isn't much fun!

Now open the w2\_02 sketch and run it. It draws exactly the same thing as the previous sketch. Three rows of three squares, each row coloured differently.

However, if we take a look at the code we'll see its doing the drawing differently. In fact, this sketch contains two different approaches to drawing the grid of squares...

You can change the method used by changing the value the variable caseNum in the setup function from 1...to 2.



Let's look at the first method, when caseNum is equal to 1, this is tested for in the draw() function by the if statement. Everything inside the opening and closing curly braces will be executed if caseNum is equal to 1, which is its value in the original sketch.

The code inside this block contains three for loops, one for each row. Let's look at the structure in more detail.

The first for loop draws the first row. The syntax looks a little confusing at first, but it's really quite simple. A for statement consists of three parts: an initialisation part, which in this case declares a variable, i, and initialises it to 0; next is a condition, which if evaluates to false will cause the loop to exit. Last is an expression that is executed each time around the loop. Everything inside the opening and closing curly brackets is executed by the for statement.

So the effect of this is to increment i by one from 0 to one less than num, which in the sketch is set to three. You can see how this value gets multiplied by 150 to generate the horizontal point for the rectangle as shown in the table.

Now let's look at the second case. If you set caseNum to 2, this will be the section of code that gets executed, due to the else if conditional.

Let's look at the use of for loops in this section.


Here we can see what is called "nested loops", that is, a loop within a loop. The outer loop increments i, the inner loop increments j. Because num is equal to three, the value of j only goes from 0 to 2, but it does this three times due to the outer loop that is changing i. I've left out the code that does the colour setting here.

This table shows all the rect calls made within the nested loops and the values of i and j. As you can see, the inner loop that increments j loops around three times because it's been executed by the outer loop that increments i.

This can take a bit of getting used to, so if you're in doubt, create a new sketch and try a few simple experiments with nested loops and print out the values of each variable using the println function.

You might be wondering why go to all this trouble to draw just nine squares? Didn't the code from the first sketch seem easier to understand. Well perhaps, but the power of using loops shifts the processing from you to the computer and makes it much easier to do more complex things.

All right, so let's take things up a notch. Open the w2\_03 sketch and press play. This sketch changes once per second, drawing a grid of squares with shadows.



Looking at the draw function you can see that, at each call of the draw function, two variables called num and gap are initialised with random values between the limits shown. num will vary between 3 and 11 and gap between 5 and 49. num represents the number of squares per row and the number of columns. gap represents the space between each square.

If there are n squares in a row, there will be n+1 gaps. To calculate the size of each square we multiply the gap size by the number of squares plus one, subtract from the width of the display window, and then divide by the number of squares.

This works both horizontally and vertically because we are placing an x by n grid of squares on a square window, so we don't need a separate calculation for the columns.

If you're up for a challenge, try changing the display window size to be 800 wide by 600 high and fit the squares uniformly on the grid.

One other thing that we need to look at: scope. Scope is another fancy computer term for the visibility of variables within a block of code. Blocks are normally defined by an opening and closing curly bracket. Looking at the nested for loops from the sketch, we can see that the variables i and j are declared inside each for loop. This means that when the execution exits from the loop at the closing curly bracket, the variable no longer exists. So if you try and print i and j outside the loop you'll get an error, with Processing telling you it can't find anything named "i" because the computer got rid of i because it thought you didn't need it as it was declared within a block.

This is actually quite sensible, because it allows you to reuse variable names in different parts of the code and not have changes to them affect other parts of the code.


If I want to access i or j outside the block of code they're declared in, I need to shift the declaration outside the loops as shown here. Now I can print the values after the loop exits. What do you think this will print? Test your answer by changing the code.

The cousin of for is while. While works similarly to for, except that it lacks the initialiser and expression components of the for loop. while keeps executing the code within its curly brackets as long as the condition is true. Here's the same nested loop code but with while loops.

So now for a challenge. Using the w2\_03 sketch as a basis, see if you can reproduce the Vera Molnar, "25 squares" artwork in Processing.

Here's some hints. Obviously you'll need to draw 25 squares. The next thing is that the squares are not perfectly aligned to the grid: they've been shifted by some small amount horizontally and vertically. If you look carefully at the image you can see darker areas where the squares overlap. Finally, while most of the squares are a light brown, occasionally some are bright red.

Rather than producing the same artwork each time the sketch is run, you might want to consider how you can change it in some way so that it still works within the rules of being "25 squares", but each time you run the program it produces a slightly different drawing - for example the number and location of red



squares might change and the shifting from the grid might vary. You'll need to use your creative judgment to decide what fits the rules of this artwork.

One function that will help you with this is the “randomSeed” function. You need to call this function once within setup. It's purpose is to seed Processing's built-in random number generator so the numbers it produces are selected from a different part of a long random sequence - you can think of it like shuffling a deck of cards.

If you call randomSeed with the same argument, you'll get the same sequence of random numbers from random(), so here I'm creating a number for the seed based on the current time of day, plus the time the program has been running in milliseconds, which is thousandths of a second. That should be enough to ensure each time you run the sketch the sequence of numbers generated by random is different.

Remember, if you need help with any Processing function, look it up in the reference, by right-clicking on the function and selecting “find in reference”.