# Week 2 - Animation, conditionals & random numbers

## Outcomes

- Understand **variables** and how to use them
- Understand **functions** and how to use them
- Using **conditional statements** to control code flow
- Using variables for animation
- Using random numbers
- Mapping values from one range to another

## Supporting code

The code for this workshop is hosted on **Github**, which is a web-based repository for hosting and versioning code.

**Download the code** and unzip it on your desktop.

> The code is also available to **view directly on Github's website**.

During this workshop session we will be using the following project directories:

```
04_using_variables/
05_animation/
06_conditionals/
07_random/
08_random_recursive_tree/
09_map_weather_api/
10_map_hsb_colours/
```

## Variables

Firstly let's take another look at variables in a bit more detail. A variable is simply a way of storing information in the computer's memory. Let's dive right in with an example...

```
var rectWidth = 5;
```

Let's break down the above statement:

1. `var` - This is how the browser knows you are 'declaring' a new variable
2. `rectWidth` - This is the name of the variable, which we can refer to later in our code. What you call a variable is up to you but there are some conventions.
3. `5` - The value which we want to store in the computer's memory

**Read more about variables in JavaScript**

# Using variables

> **Supporting Code**
> The code to support this section is located in the following directory and is available to view on **Github**:
>
> ```
> /04_using_variables/
> ```

Now that our variable `rectWidth` is stored in memory, we can access it using its name to return the stored value.

```
var rectWidth = 5;
var rectHeight = 7;
var rectArea = rectWidth * rectHeight;
console.log(rectArea); // This will write 35 to the console.
```

In this example, a new variable `rectHeight` is declared and assigned a value of 7. On the third line both the previous variable values are retrieved from memory and multiplied using the multiply operator (`*`). This is immediately stored in the `rectArea` variable before finally being logged to the console.

Here is what happens line by line:

1. Store the number 5 in a variable named `rectWidth`
2. Store the number 7 in a variable named `rectHeight`
3. Multiply the values in `rectWidth` and `rectHeight`, storing the result in a variable named `rectArea`
4. Log the value of `rectArea`

# Exercise

- Add the `04_using_variables` directory to Atom
- Open `index.html` in a browser
- Open and look at the console in the browser's developer tools
- Remove the comments at the beginning of line 20 and reload your browser

# Animation using variables

The code to support this section is located in the following directory and is available to view on **Github**:

```
/05_animation/
```

In this exercise a variable will is used to store, retrieve and increase a value. This value will represent the position of a shape drawn to the canvas.

Here is a portion of the code extracted from the provided example:

```
var positionX = 0;

/*
[code excluded]
*/

function draw(){
  // Set the background to black every frame
  background(0);

  // Draw a rectangle that moves along the X axis
  rect(positionX, height/2, 10, 10);

  // Increase the value stored in positionX
  positionX = positionX + 1;
}
```

As you can see a variable called `positionX` is declared and assigned a value of 0. Importantly this variable is declared *outside* of the function where it is later used. The variable is declared in the **global scope** (more on this later) making it accessible throughout the entire application (i.e. globally).

# Exercise

- Add the `05_animation` directory to Atom

- Open `index.html` in a browser
- Use the conditional `if` statement to reset the square to position 0.
- Increase the speed of the rectangle
- Move the rectangle on the X and Y axis

# Conditionals

The code to support this section is located in the following directory and is available to view on **Github**:

```
/06_conditionals/
```

A conditional statement is used to control which code is executed based on certain pre-determined conditions. This process is one method of **controlling the flow** of our application.

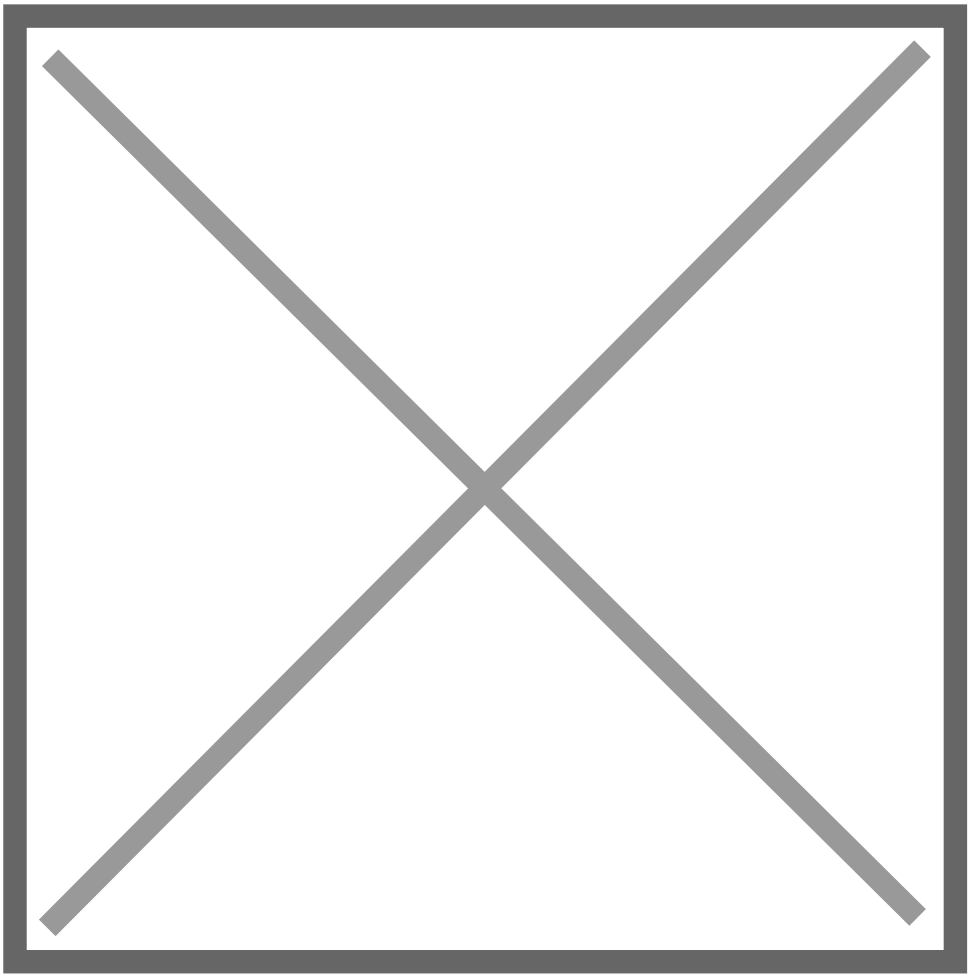
## If statements

Conditional statements are written in code using the `if` keyword. In fact, conditional statements are often referred to as *if statements*. Below is an example of how a conditional statement is formed using the `if` keyword:

```
if (condition) {
  // code that runs if the condition is true
}
```

By replacing the `condition` above with other statements we can start to control what parts of our code are executed under which conditions.

You can think of this as a very simple flow diagram or decision tree. If `condition A` is TRUE then the code block runs, however if it's FALSE the code is ignored.



## Is the statement true or false?

When writing a condition, commonly known as a conditional statement, the truth of the statement is being evaluated or checked. In the following examples this happens by comparing two values.

These values can be variables, **literal values** or a combination of the two.

> **Literal values**
> Literal values are those that we write in our code *literally.*As opposed to variables that can change, these values are written explicitly in our code and **do not change**. Here are some examples:
>
> ```
> "Hello, World"
> 12
> 3.141592
> ```

Here are some practical examples of `if` statements that use both variables and literal values. Between each set of brackets is a statement comparing two values. Those comparisons will return a value of **true** or **false**, which determines if the code within should be **executed** or **ignored**.

```
if( userName == "bob" ){
   // Any code in here will run when userName is equal to ('==') "bob"
}

if( durationHours > 12 ){
  // Any code in here will run when durationHours is greater than (`>`) 12
}

if( rectArea <= 35 ){
  // Any code in here will run when rectArea is less than OR equal to ('<=') 35
}
```

## Comparison operators

In conditional statements, a comparison operator sits between the two values and is used to determine whether the statement is true or false. Below is a list of conditional statements using different comparison operators.

| | |
|---|---|
| `A == B` | A equal to B |
| `A != B` | A is not equal to B |
| `A > B` | A is greater than B |
| `A < B` | A is less than B |
| `A >= B` | A is greater than or equal to B |
| `A <= B` | A is less than or equal to B |

If the statement is true then the code within the conditional will run. Here are some more practical examples:

```
value1 == value2
userName == "bob"
playerScore >= 10
"west" == windDirection
juneTemperature > mayTemperature
```

Let's break down one of the above conditions:

1. `userName`
   A variable – as the word 'variable' suggests, we expect it may change.

2. `==`
   A comparison operator checking for **equality** – checks if the value on the left is equal to the value on the right.
3. `"bob"`
   A string literal – written explicitly and therefore will not change.

Since variables can change value throughout the execution of code, the comparison to a static value causes code to run only during particular conditions.

If variables are named well you can start to read through the logical steps of your application by reading the code as human language:

```
if the userName is equal to "bob"
  then do something
```

> **Double (==) and single (=) equals signs**
> Always be sure to use the double equals sign in conditional if statements. Using the single equals sign will change the value stored inside the variable.

## Exercise

- Add the `06_conditionals` directory to Atom
- Open `index.html` in a browser
- Modify the code inside the first conditional to make the ball bounce off the right side of the canvas
- Use another conditional to make the ball bounce off both sides of the canvas
- Change the colour, size, speed of the ball when it bounces off the wall
- Move up and down instead of left and right

# Using random numbers

The code to support this section is located in the following directory and is available to view on **Github**:

```
/07_random/
```

Most programming languages provide functions for generating random numbers. This can be very useful in providing some variations to deterministic behaviour of code.

In p5.js there is a function for generating a random number between a minimum and maximum value:

```
random(min, max);
```

The `min` and `max` arguments set the minimum and maximum values that can be returned from that function.

```
random(0, 10);
random(120, 180)
random(15, 22);
```

You can also use a variable as one of the arguments:

```
random(0, width);
random(0, height);
```

The `random()` function can be used to set properties of shapes in our sketch such as position, size or colour.

In the following example the `positionX` and `positionY` variables are assigned values that are half of the width and half of the height of the canvas respectively. This will place the ellipse in the centre of the canvas when the code runs.

```
var positionX;
var positionY;

function setup() {
  createCanvas(800, 450);
  // Assign a value to the variables
  positionX = width/2;
  positionY = height/2;
}

function draw() {
  // Use the value within the variables.
  ellipse(positionX, positionY, 10, 10);
}
```

Here is an example of how to use the random function to change the starting position of the ellipse to a random position on the canvas on every execution of the code.

```
var positionX;
var positionY;

function setup() {
```

```
  createCanvas(800, 450);
  // Assign a value to the variables
  positionX = random(0, width); // Random number between 0 & 800
  positionY = random(0, height); // Random number between 0 & 450
}

function draw() {
  // Use the value within the variables.
  ellipse(positionX, positionY, 10, 10);
}
```

> **The Nature of Code**
> For an in-depth look at how random numbers relate to other programming concepts such as probability, evolutionary programming and the 1982 sci-fi classic **Tron**, take a look at Daniel Shiffman's free online book **The Nature of Code**.

# Exercise

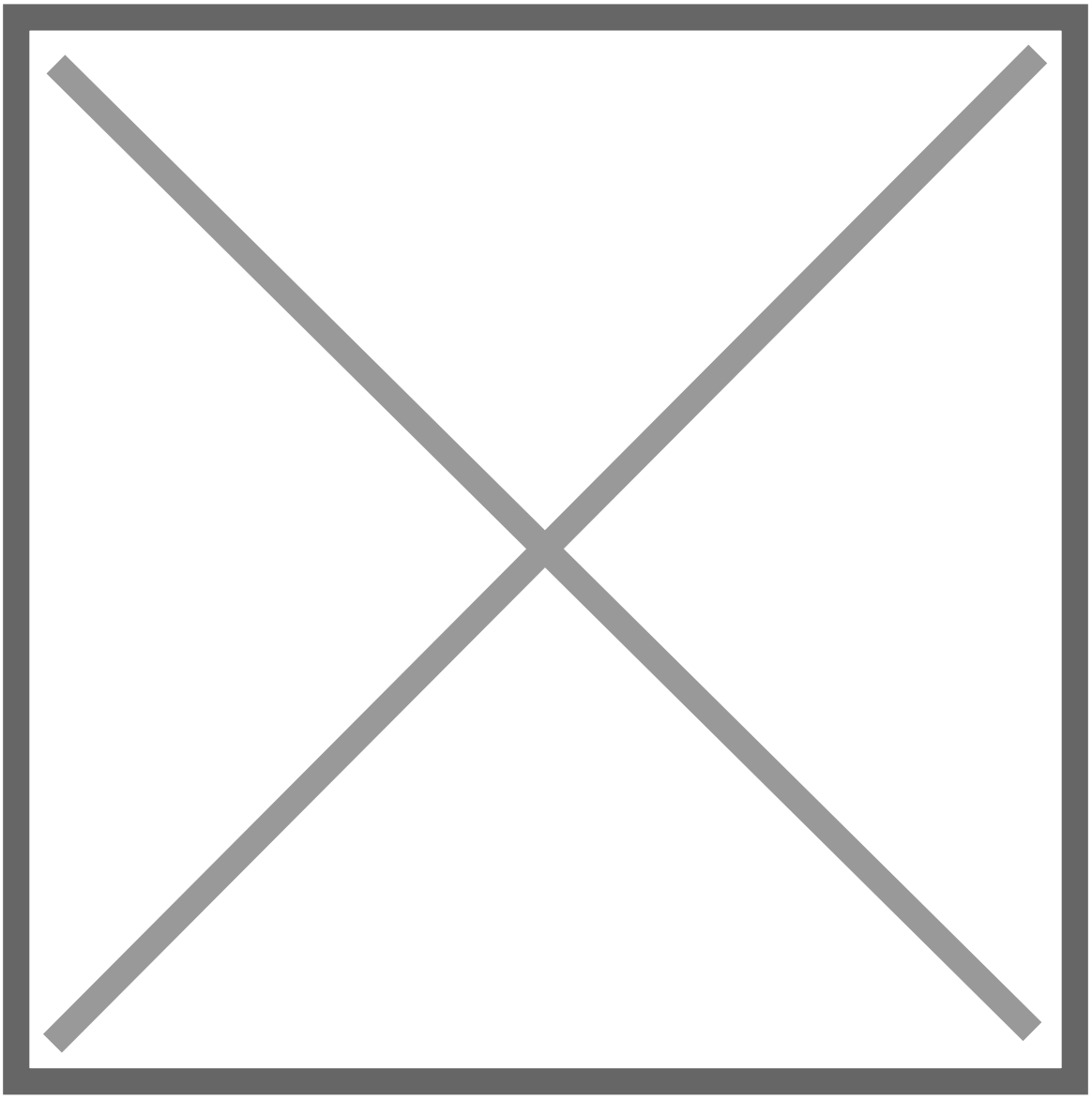
- Add the `07_random` directory to Atom
- Open `index.html` in a browser
- Change the X and Y positions of the ellipse using `random()` on every frame
- Change another feature of the shape with random (size, colour, etc)

# Randomness and probability

The code to support this section is located in the following directory and is available to view on **Github**:

```
/08_random_recursive_tree/
```

- Add the `08_random_recursive_tree` directory to Atom
- Open `index.html` in a browser and you will see something similar to this:



This is an example of using randomness and probability to produce organic forms. Take a look through the code and you will see some lines such as this:

```
// Create a random numbers between 0 and 1
var r = random(0, 1.0);

// 98% chance this will happen
if (r > 0.02) {
  [code excluded here]
}
// 2% chance this will happen
else {
  [code excluded here]
}
```

You can see that by using random numbers and conditional statements you can quite easily create systems that have interesting and unexpected results within the limits of probability.

This code also uses a very powerful technique called recursion, which is beyond the scope of this workshop. Essentially the code is self-referential and therefore within very few lines of code can create complex outputs.

# Mapping values

The code to support this section is located in the following directory and is available to view on **Github**:

```
/09_map_weather_api/
```

A common programming task – particularly when visualising information – is to take a value that is changing within one range and *mapping* that onto a different range.

As an example, let's think about visualising the current temperature (a changing value) by drawing a thermostat.


We know that the value is going to be in this approximate range of 0 to 50 °C and the size of the red thermostat indicator is a shape with a height between 0 and 200 pixels:

| | MIN | MAX |
|---|---|---|
| °C | 0 | 50 |
| pixels | 0 | 200 |

Let's assume we have retrieved the current temperature in degrees centigrade, for example through a weather API.

If the temperature is 50°C, the height of the red bar would be 200 pixels; if the temperature is 0°C, the height would be 0 pixels; and if the temperature is 25°C (half way point of the range), the height would be 100 pixels (half the height).

| Current Temp (°C) | Height (pixels) |
|---|---|
| 0 | 0 |
| 50 | 200 |
| 25 | 100 |

| Current Temp (°C) | Height (pixels) |
| --- | --- |
| 10 | 40 |
| 35 | 140 |

## Using the map function

Within p5.js there the **map function** performs the calculations that translates one range onto another. The `map()` function takes 5 arguments:

```
map(value, fromMin, fromMax, toMin, toMax);
```

So using the example of the thermostat, we would convert the current temperature stored in a variable called `temperature` using the following:

```
map(temperature, 0, 50, 0, 200);
```

And here are some examples from above using literal integer values:

```
map( 25, 0, 50, 0, 200 ) // returns 100
map( 10, 0, 50, 0, 200 ) // returns 40
map( 35, 0, 50, 0, 200 ) // returns 140
```

## Exercise

- Add the `09_map_weather_api` directory to Atom
- Open `index.html` in a browser
- Look through the code and find where the `map()` function is used
- Change the city in the preload function to see the API results from other places

# HSB Colour

The code to support this section is located in the following directory and is available to view on **Github**:

```
/10_map_hsb_colours/
```

Using the RGB colour space we can produce as the specific colours we need. However, in order to manipulate or generate colours, the RGB colour space doesn't offer the best tools. For this we can use the HSB colour space or Hue, Saturation and Brightness. It is sometimes also known as as HSL (lightness) or HSV (value).

Within this model the **hue** defines the colour we see, which is the wavelength of light being produced. The **saturation** defines how intense or vivid the colour is. The way the colour is desaturated is by the addition of grey: 100% saturation means there is no grey and 0% saturation will result in a medium grey. The **brightness** determines the amount of black or white that's mixed with the hue.

Here are the RGB and HSB colour spaces visualised:

rgb-hsb.png

## Changing colour mode

In p5.js you can change the colour space from RGB to HSB using the following.

```
colorMode(HSB);
```

The `colorMode` function can also take 3 more arguments:

```
colorMode(HSB, 360, 100, 100);
```

These last 3 arguments represent the range of values we can pass as arguments into the colour functions such as `fill()` and `stroke()`.

In RGB colour mode, the range is by default:

| Red | Green | Blue |
|---|---|---|
| 0 - 255 | 0 - 255 | 0 - 255 |

But in HSB mode, the hue is usually between 0 and 360 whilst the saturation and brightness are between 0 and 100.

| Hue | Saturation | Brightness |
|---|---|---|
| 0 - 360 | 0 - 100 | 0 - 100 |

The saturation and brightness are essentially represented as a percentage (0 to 100%) of their most extreme condition, which is the least saturated and the most bright.

But why is the hue value between 0 and 360? As mentioned the HSB colour is visualised as a cylinder (or sometimes as a cone) and the hue is represented as the perimeter of the circle that sits at the top of the 3D shape. Therefore the 360 is the angle in degrees around that circle.

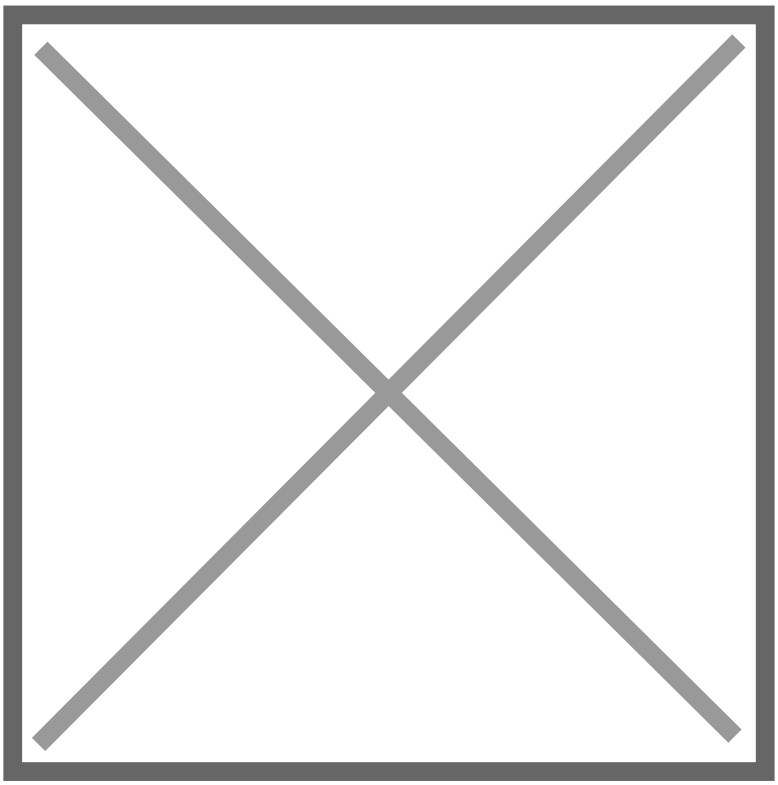


Image credit: **www.runemadsen.com**

# Exercise

Using the HSB colour space we can create easily create colour schemes that have a mathematical relationship to each other. A simple example is choosing a particular hue and saturation and then adjusting the brightness. However you can also choose selections of hue based on their relationship around the 360 degrees of the colour wheel.

| | |
|---|---|
| analogous-5905a98134b0e87c7822f38cf9af3d62_large.jpg<br>**Analogous** | complementaries-41a71e8df01c8b7e659808b1d03289f0_lar<br>**Complementary** |
| triadic-9adb1731f0659e77584becced63e35ef_large.jpg<br>**Triadic** | tetradic-768b73622eb3aec919d28e8edcad2f51_large.jpg<br>**Tetradic** |

All of these examples are from the **Rune Madesen's lecture on colour** as part of his Printing Code module at ITP. The online resources from this are extremely useful.

In the provided example, the `mouseX` value is being mapped from one range (0 to `width`) onto another (0 to 360):

| | MIN | MAX |
|---|---|---|
| width of canvas | 0 | 50 |
| degrees of colour wheel | 0 | 360 |

Therefore as the mouse moves across the canvas the mapped value travels between 0 and 360. This is then used to set the **hue** of the fill colour showing the full spectrum of colour.

```
var colour = map(mouseX, 0, width, 0, 360);
var columnWidth = width/3;

fill(colour, 100, 100);
rect(columnWidth*0, 0, columnWidth, height);

fill(colour, 80, 70);
rect(columnWidth*1, 0, columnWidth, height);

fill(colour, 60, 40);
rect(columnWidth*2, 0, columnWidth, height);
```

- Add the `10_map_hsb_colours` directory to Atom
- Open `index.html` in a browser
- Explore different values for brightness and saturation
- Create colour schemes with hues that have are related on the colour wheel, e.g. analogous, triadic, etc.

# Functions

Functions are used to define a process that can be constructed of one or more lines of code. They are often used to organise and structure code by the intended outcome or behaviour.

Here are a few benefits to using functions:

1. Keep code organised
2. Make code easily reusable
3. Breaking down a task into smaller pieces (decomposition)
4. Making problems in the code easier to identify and troubleshoot (seperation of concerns)

## Using functions

Making use of functions is broken down into two parts. First, the function behaviour needs to be defined, i.e. the code needs to be written. Secondly, the function needs to be called (also known as 'executed').

### Define the function behaviour

Below are 4 lines of code contained within a function that perform the task of calculating the area of a shape. This is where the function is being defined.

```
function calculateArea() {
  var width = 5;
  var height = 7;
  var area = width * height;
  console.log(area);
}
```

Let's break down the unfamiliar parts of the above code:

1. `function`
   This is how the browser knows you are declaring a new function.
2. `calculateArea()`
   'calculateArea' is the name of the function, which we can use to refer to later in our code.
   What you call a function is up to you but there are some conventions.
3. `{ }`
   These are curly brackets or curly braces. They start and end the content of the function.
   All code written between these two brackets is the behaviour of the function.

## Call the function

The above code will do nothing until we call the function elsewhere in our code.

```
calculateArea(); // Logs 35
```

# Function parameters

A common use of a function is to make our code more reusable. One way of making our functions more reusable is by adding parameters.

```
function calculateArea( width, height ) {
  var area = width * height;
  console.log(area);
}
```

# Assignment

## Part 1

Create a sketch that includes:

- one or more elements that changes over time.
- one or more elements that is controlled by mouse or keyboard

- one or more element that is random() in nature

Work can again be submitted using Codepen. Here is the URL for the p5.js template:

**http://codepen.io/pen?template=zKLpKw**

**Please submit the Codepen URL the day before our next workshop.**

And here is a short guide on using Codepen:

Codepen - Create Pen from template

# Part 2

When you submit your URL I would like you to also submit a question about what we've been covering (or have missed) over the last two weeks. For example:

- What does a certain error message mean?
- How do I create a colour with an alpha channel?
- Are there any other colorModes?
- What is the highest framerate?

---

Revision #40
Created 30 October 2016 18:26:24
Updated 20 August 2018 11:13:01