**JavaScript**

**Fundamentals 2**

**Use Strict Mode:**

'use strict';

This must be the first line of code in a script.

Strict mode makes it easier to avoid accidental errors or introduce bugs into our code.

Strict mode:

1. Prevents us from performing certain tasks.
2. Will display visible errors.

**Functions**

Graphical user interface, application

Description automatically generated

A function can hold one or more complete lines of code.

*function* logger() {

}

This is an example of a function. All of the code within the { } is called the function buddy. It is the code that will be executed when the function is called.

*function* logger() {

console.log('Dylan is cute.');

}

logger();

And this is how you call your function. Each time we call the function, the code that is in the function will get executed. In this instance, we would get ‘Dylan is cute.’ Printed to the console one time. If we continued calling the function then we would get it multiple times.

Functions can also receive data and return data back to us. Functions are a bit like ‘machines’. This diagram may look a little confusing but it is actually quite simple.

1. Step 1, we declare our function by giving it the identifier ‘getGreeting()’.
2. We then say how we would like our function to operate when it is called. We put this within the curly braces (number 3.).
3. Step 2, we call the function by writing ‘getGreeting(); ‘.
4. Step 3, the function is called and the code within the function is used.
5. The code within the function IS NOT used unless it is called. That is why it is not step 2.

Graphical user interface, text, application

Description automatically generated

*function* fruitProcessor(*apples*, *oranges*) {

console.log(*apples*, *oranges*);

*const* juice = `Juice with ${*apples*} apples and ${*oranges*} oranges.`;

return juice;

}

fruitProcessor(1, 3);

Here we have called our function ‘fruitProcessor’. And we are using it to display how many apples and oranges we need for our juice.

We declare a string called ‘const juice’.

Later we call our ‘fruitProcessor’ function by inputting how many apples and oranges we want to use.

In order for us to print something meaningful to the console, we need to do this;

*const* appleJuice = fruitProcessor(1, 3);

console.log(appleJuice);

This will return to the console: “Juice with 1 apples and 3 oranges.”

We created a variable ‘appleJuice’. This variable then calls our ‘fruitProcessor’ function with two arguments; 1 and 3. Looking back where we defined our function, the numbers correspond to the amount of apples and oranges.

When we use the ‘return’ command we then need to store the information somewhere. We create the appleJuice variable as a way to store this information.

*const* appleOrangeJuice = fruitProcessor(2, 4);

console.log(appleOrangeJuice);

We can call the function again with different variables.

**Parameters and Arguments**

Parameters allow functions to accept input(s) and perform a task using the input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

Diagram

Description automatically generated

In this diagram, calculateArea() is calculating the area of a rectangle, based upon two inputs, ‘width’ and ‘height’. The parameters are specified between the parenthesis as ‘width’ and ‘height’, and inside the function body, they act just like regular variables.

‘width’ and ‘height’ act as placeholders for values that will be multiplied together.

Diagram

Description automatically generatedWhen calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called ‘arguments’. Arguments can be passed to the function as values or variables.

In this example, the number 10 is passed as the width and 6 is passed as the height. The order in which arguments are passed and assigned follows the order that the parameters are declared.

Timeline

Description automatically generated

The variables rectWidth and rectHeight are initialised with the values for the height and width of a rectangle before being used in the function call.

By using parameters, calculateArea() can be reused to compute the area of any rectangle!

Functions are a powerful tool in computer programming so let’s practice creating and calling functions with parameters.

Here is another example of some code:

function sayThanks(name) {

console.log('Thank you for your purchase '+ name + '! We appreciate your business.');

}

sayThanks('Cole');

In this code the console logs the name Cole with a thank you message.

The function identifier is ‘sayThanks’.

The argument is simply ‘name’.

We call the argument ‘sayThanks’ and input the name ‘Cole’. Hence the name is implemented into our code and console output.

If we did not put in ‘Cole’ and left it empty, the name would be returned as ‘undefined’.

**Default Parameters**

In ES6 is the ability to use default parameters. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is ‘undefined’ when called.

function greeting (name = 'stranger') {  
  console.log(`Hello, ${name}!`)  
}  
  
greeting('Nick') // Output: Hello, Nick!  
greeting() // Output: Hello, stranger!

For example:

1. We used the = operator to assign the parameter name a default value of ‘stranger’. This is useful to have in case we ever want to include a non-personalised default greeting.
2. When the code calls greeting(‘Nick’) the value of the argument is passed in and , ‘Nick’, will override the default parameter of ‘stranger’ to log ‘Hello, Nick!’ to the console.
3. When there isn’t an argument passed into greeting(), the default value of ‘stranger’ is used, and ‘Hello, stranger!’ is logged to the console.

By using a default parameter, we account for situations when an argument is not passed into a function that is expecting an argument.

**Return**

When a function is called, the computer will run through the function’s code and evaluate the result of calling the function. By default the resulting value is undefined.

function rectangleArea(width, height) {  
  let area = width \* height;  
}  
console.log(rectangleArea(5, 7)) // Prints undefined

In this code example we defined our function to calculate the area of a width and height parameter. Then rectangleArea() is invoked with the arguments ‘5’ and ‘7’. But when we went to print the results we got undefined. The function worked fine and the computer did calculate the area as 35, but we did not capture it. So how can we do this? We use the keyword ‘return’.

Graphical user interface

Description automatically generated with low confidence

In order to pass back information from the function call, we need to use a return statement. To create a return statement, we use the return keyword followed by the value that we wish to return. If the value is omitted, ‘undefined’ is returned instead.

When a return statement is used in a function body, the execution of the function is stopped and the code that follows will not be executed.

function rectangleArea(width, height) {  
  if (width < 0 || height < 0) {  
    return 'You need positive integers to calculate area!';  
  }  
  return width \* height;  
}

If an argument for ‘width’ or ‘height’ is less than 0 then rectangleArea() will return ‘You need positive integers to calculate area!’. The second return statement ‘width \* height’ will not run.

The return keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

function monitorCount(rows, columns){

return rows \* columns;

};

const numOfMonitors = monitorCount(5, 4);

console.log(numOfMonitors);

In this code we are trying to work out how many computer monitors we need to purchase. First, we have declared a function that has two parameters. The first parameter is rows and the second is columns. Next we have used the return keyword to return ‘rows \* column’s.

The function is now defined.

We can now declare a variable named ‘numOfMonitors’. If we input ‘4’ and ‘5’ we will return 20 to the console.

**Helper Functions**

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as a helper function. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like this:

function multiplyByNineFifths(number) {  
  return number \* (9/5);  
};  
  
function getFahrenheit(celsius) {  
  return multiplyByNineFifths(celsius) + 32;  
};  
  
getFahrenheit(15); // Returns 59

In this example:

* getFahrenheit() is called and 15 is passed as an argument: getFahrenheit(15);
* The code block inside of getFahrenheit() calls multiplyByNineFifths() takes the argument of 15 for the number parameter.
* The code block inside of multiplyByNineFifths() function multiples 15 by (9/4) which evaluates to 27.
* 27 is returned back to the function call in getFahrenheit().
* getFahrenheit() continues to execute. It adds 32 to 27 which evaluates to 59.
* Finally, 59 is returned back to the function call getFahrenheit(15).

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficult tasks and break them into smaller more manageable pieces.

**Function Expressions**

Diagram, timeline

Description automatically generated with medium confidenceAnother way to define a function is to use a function expression. To define a function inside an expression, we can use the ‘function’ keyword. In a function expression, the function name is usually omitted. A function with no name is called an anonymous function. A function expression is often stored in a variable in order to refer to it.

To declare a function expression:

1. Declare a variable to make the variable’s name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use ‘const’ as the keyword to declare the variable.
2. Assign as that variable’s value an anonymous function created by using the ‘function’ keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

variableName(argument1, argument2)

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

Function expressions can not be called before they have been defined.

**Function Declaration**

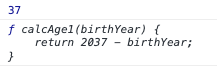
Function declarations can be called before they are defined. Here we have called our function (calcAge1) BEFORE it has been defined. This does not work for function expressions.

*const* age1 = calcAge1(1991);

*function* calcAge1(*birthYear*) {

return 2037 - *birthYear*;

}

We want a persons birthyear to be an input for the function. We have created a function expression that should perform a calculation but what are the results: It is important to note that the function will only execute correctly if it is given some parameters. This is why we have two different results here!

We can think of ‘birthYear’ here as being similar to a ‘local variable’ that only exists within the function.

The function will not execute unless it is called.

//Function Declaration

*function* calcAge1(*birthYear*) {

return 2037 - *birthYear*;

}

*const* age1 = calcAge1(1991);

//Function Expression

*const* calcAge2 = *function* (*birthYear*) {

return 2037 - *birthYear*;

}

*const* age2 = calcAge2(1991);

console.log(age1, age2);

Here is a **function expression**. It is a function without a name. Think of it as not have a ‘declared’ name; hence it is an expression.

A function expression can be stored as a variable. The variable can then be used as the function.

This is a function expression. Remember that an expression will produce a value. Here we have stored our value in calcAge2.

We call this function in the same way;

**Arrow Functions**

ES6 introduced arrow function syntax, a shorter way to write functions by using the special “fat arrow” () => notation.

Arrow functions remove the need to type out the keyword function every time you need to create a function. Instead, you first include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { } like this:

const rectangleArea = (width, height) => {  
  let area = width \* height;  
  return area;  
};

It’s important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

*const* calcAge3 = *birthYear* *=>* 2037 - *birthYear*;

*const* age3 = calcAge3(1991);

console.log(age3);

First the function have been defined and saved to a variable. Next the function has been called by creating the variable ‘age3’. Finally we have logged the result of the function being called to the console.

**Concise Body Arrow Functions**

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as concise body.

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.

Graphical user interface, application

Description automatically generated

1. Graphical user interface, text

   Description automatically generatedA function body composed of a single line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow => and the return keyword can be removed. This is referred to as an implicit return.

So if we have a function:

const squareNum = (num) => {  
  return num \* num;  
};

We can refactor the function to:

const squareNum = num => num \* num;

Notice the following changes:

* The parentheses around ‘num’ have been removed, since it has a single parameter.
* The curly braces { } have been removed since the function consists of a single-line block.
* The return keyword has been removed since the function consists of a single-line block.

Here is another example of a function. Let’s assume that we want to calculate how many years we have left until retirement.

*const* yearsUntilRetirement = *birthYear* *=>* {

*const* age = 2037 - *birthYear*;

*const* retirement = 65 - age;

return retirement;

}

console.log(yearsUntilRetirement(1991));

First, we have declared the variable ‘yearsUntilRetirement.’. Then we have assigned it the following parameter; birthYear.

Create a variable to determine age. This assumes that the year is currently 2037.

If we assume that the age of retirement is 65, we then create a variable that takes our variable age from 65.

After the calculation has been completed we want the function to return our calculated value for retirement. Hence; return retirement.

But what if we had multiple variables that we needed to take into account? Perhaps in our previous example we want to set the birth year and the first name as our possible parameters.

We use parentheses:

*const* yearsUntilRetirement = (*birthYear*, *firstName*) *=>*

Now we have a variable that has a function with two parameters; birth year and first name.

*const* yearsUntilRetirement = (*birthYear*, *firstName*) *=>* {

*const* age = 2037 - *birthYear*;

*const* retirement = 65 - age;

//return retirement;

return `${*firstName*} retires in ${retirement} years.`

}

console.log(yearsUntilRetirement(1991, 'Simon'));

And here is our completed code. Notice that when we return this function we have used a template literal to create our string. When we call our function it is important to define both parameters, in this case we have selected 1991 and Simon. These parameters will now be used by the function we created earlier.

**Functions Calling other Functions!**

In JavaScript we often need one function to call another function. Lets take our food processor analogy from earlier and develop it further.

*function* cutFruitPieces(*fruit*) {

return *fruit* \* 4;

}

First of all, lets create a function that multiplies the amount of fruit we have by 4. Think of this a bit like having an orange and putting it into a machine that will cut it into 4 segments.

So if this is our blender, how do we implement it into another function?

*function* fruitProcessor(*apples*, *oranges*) {

*const* applePieces = cutFruitPieces(*apples*);

*const* orangePieces = cutFruitPieces(*oranges*);

*const* juice = `Juice with ${*applePieces*} apples and ${*orangePieces*} oranges.`

return juice;

};

fruitProcessor(2, 3);

console.log(fruitProcessor(2, 3));

Like this. Notice that we call the function by imputing our parameters, in this case 2 and 3. The 2 and 3 correspond to the number of apples and oranges we have.

The fruit processor function creates the variable applePieces, which takes our number of apples and calls our cutFruitPieces function which multiplies by 4. The same is also true for the oranges.

Then a variable is created called ‘juice’ that prints how many apples and oranges are required for our drink. Finally, so that the code will be executed, return is used. The console log will return; ‘Juice with 8 apples and 12 oranges.’

**Review Functions**

* A function is a reusable block of code that groups together a sequence of statements to perform a specific task.
* A picture containing graphical user interface

  Description automatically generatedA function declaration:
* Diagram

  Description automatically generatedA parameter is a named variable inside a function’s block which will be assigned the value of the argument passed in when the function is invoked:
* Diagram

  Description automatically generatedTo call a function in your code:
* ES6 introduces new ways of handling arbitrary parameters through default parameters which allow us to assign a default value to a parameter in case no argument is passed into the function.
* To return a value from a function, we use a return statement.
* To define a function using function expressions:

Timeline

Description automatically generated

* To define a function using arrow function notation:Timeline

  Description automatically generated
* Function definition can be made concise using concise arrow notation:Graphical user interface

  Description automatically generated

Be aware of the differences between function expressions, arrow functions, and function declarations. As you program more in JavaScript, you’ll see a wide variety of how these function types are used.

Function Expression: A function value stored in a variable.

*const* yearsUntilRetirement = *function* (*birthYear*, *firstName*) {

Function Declaration: Function can be used before its declared.

*function* yearsUntilRetirement (*birthYear*, *firstName*)

Arrow Function: Quick one-line functions.

*const* yearsUntilRetirement = (*birthYear*, *firstName*) *=>*

**Parameters can be shared across multiple functions**

*const* calcAge = *function*(*birthYear*) {

return 2037 - *birthYear*;

}

*const* yearsUntilRetirement = (*birthYear*, *firstName*) *=>* {

*const* age = 2037 - *birthYear*;

Here we have two different functions. But they both use the birthyear parameter. We can do this because they are two different machines. They will only accept our parameters when we call the function. Parameters are like a local variable to each function.

**If/Else and Returns**

*const* yearsUntilRetirement = (*birthYear*, *firstName*) *=>* {

*const* age = calcAge(*birthYear*);

*const* retirement = 65 - age;

if(retirement > 0) {

return retirement;

console.log(`${*firstName*} retires in ${retirement} years.`)

} else {

return -1;

console.log(`${*firstName*} has already retired.`)

}

In this code the console.log’s will not get printed to the console. This is because they are after the return keyword. The function has already been returned and no further code will be executed. In order to work, it must be put before the return keyword. Remember that the return statement outputs a value from the function and terminates the execution.

**Calling a Function**

*const* age = calcAge(1991, 'Jonas');

Creating a new variable is one method of ‘calling’ a function. We could console.log it too. We call the function by naming it followed by parentheses. Within the parentheses we have our arguments. These are the values that will replace the parameters in the function.

**Arrays**

Perhaps we wanted to store multiple variables:

*const* pet1 = 'Dylan';

*const* pet2 = 'Ham';

*const* pet3 = 'Randy';

Here are some names of pets. But what if we were a pet store and we had 100’s of pets? We would use a data structure to do this. An array is a bit like a big container/database in which we can add variables and then later reference them.

A picture containing graphical user interface

Description automatically generated

The two most important data structures in JavaScript are arrays and objects.

**Method 1:**

**The Literal Syntax; [ ]**

*const* pets = ['Dylan', 'Ham', 'Randy', 'Billie', 'Quinn'];

To create an array we do it like this. In the console, our array would look like this:

**Method 2:**

*const* years = new *Array*(1991, 1984, 2008, 2020);

This method is different because we used an array function.

The most common method is the literal syntax.

**Data Extraction**

We have stored data in our array using literal syntax – but how do we extract the data?

console.log(pets[3]);

This would return ‘Billie’ as she is number 3.

Dylan = 0, Ham = 1, Randy = 2 and Billie = 3.

**Array Length**

We can determine how many objects/data points there are within an array.

console.log(pets.length);

This is how we do it, with the .length property.

**Last Element of an Array**

We can use the .length property to get the last element of an array.

Here is our array:

*const* pets = ['Dylan', 'Ham', 'Randy', 'Billie', 'Quinn'];

Remember that the array is 0 based. Therefore Dylan is 0 and Quinn is **4**.

If we use the .length array the number 5 will be returned to the console. Why? Because it is the number of all our data points. That means we need to perform a small calculation on our .length property.

console.log(pets[pets.length - 1]);

And this is the method. We use the pets.length expression. An expression produces a value. In this case it would be Quinn.

**Replace/Mutate Array**

Sometimes we need to delete data or change data within our array.

pets[2] = 'Amber';

This is how we do it. The position is the number within our square [ ] brackets. This means we will change the number at position 2 to Amber.

**Note**

We declared a ‘const’ variable for pets. How are we able to change the values?

Only primitive values are immutable. Primitive values are ‘strings’, ‘numbers’, ‘booleans’ etc.

An array is not a primitive value. While an array does contain primitive values – in this case strings. An array itself is *not* a primitive value, therefore it is mutable.

**Different Types of Values**

Arrays can hold different types of primitive values and they are not constrained to holding only one type.

*const* firstName = 'Jonas';

*const* jonas = [firstName, 'Schmedtmann', 2037 - 1991];

Notice that within arrays there is flexibility regarding how we input/store data.

Firstly we created a variable called ‘firstName’. We then implemented that variable within our array.

Secondly we have also added an expression to our array, which will produce a value. We could even put an array inside an array. This convenient because we don’t need to create one variable per data point.

*const* jonas = [firstName, 'Schmedtmann', 2037 - 1991, 'teacher', pets];

Notice how we have implemented our data point from earlier? 

**Arrays and Functions**

*const* calcAge = *function* (*birthYear*) {

return 2023 - *birthYear*;

};

*const* years = [1990, 1967, 2002, 2010, 2018];

Here we have created a function; calcAge. We also have an array; ‘years’. But what if we wanted to calculate age for individual elements within our array?

Example 1:

*const* age1 = calcAge(years[0]);

*const* age2 = calcAge(years[1]);

*const* age3 = calcAge(years[years.length - 1]);

console.log(age1, age2, age3);

Here is how we do it.

We could do it another way;

Example 2:

*const* ages = [calcAge(years[0]), calcAge(years[1]), calcAge(years[years.length - 1])];

console.log(ages);

Remember that we can use expressions within our array. This way we can abbreviate example 1 into example 2. Remember that expressions produce values.

**Basic Array Operations**

**Push Method**

Adds elements to the end of an array.

*const* friends = ['Michale', 'Steven', 'Peter'];

friends.push('Jay');

Here we have added ‘Jay’ into our friends array.

*const* friends = ['Michale', 'Steven', 'Peter'];

*const* newLength = friends.push('Jay');

console.log(friends);

console.log(newLength);

**Unshift Method**

We can add elements to the beginning of an array. We use the unshift method.

friends.unshift('John');

**Remove Elements**

**Pop Method**

Removes the last element of an Array.

friends.pop();

No argument needs to be presented in the parentheses as we are just removing the last element in the array.

friends.pop();

*const* popped = friends.pop();

console.log(popped);

Table

Description automatically generated

The first pop takes out ‘Jay’. There is then a variable ‘const popped’ that does the same thing. We can use the console command to see which name we have removed from the array; ‘Peter’.

**Shift Method**

The shift method removes elements from the beginning of the array.

friends.shift();

No arguments are necessary as by default it will remove any element at the beginning of the array.

**.indexOf**

Index of is useful as it can tell us the position of an element within an array.

console.log(friends.indexOf('Steven'));

And it would return position 2. This is because Steven is in position 2. A picture containing graphical user interface

Description automatically generated

If we search for an element that does not exist in the array. We will get ‘-1’ returned.

**Includes**

ES6 version features the includes function. This will tell you if an element is present in an array or not;

console.log(friends.includes('Steven'));

console.log(friends.includes('Bob'));

For our first example ‘true’ will be returned to the console. For the second; ‘false’.

This tests with strict equality ‘===’ which means it does not do automatic type coercion ‘23’ the string is not equal to 23 the number.

**Objects**

Up to this point, we have used arrays to store multiple data values. We can’t list information that gives our data meaning in an array.

//An Array:

*const* jonasArray = [

'Jonas',

'Schmedtmann',

2037-1991,

'Teacher',

['Michael', 'Peter', 'Steven']

] ;

//An Object:

*const* jonas = {

firstName: 'Jonas',

lastName: 'Schmedtmann',

age: 2037-1991,

job: 'Teacher',

friends: ['Michael', 'Peter', 'Steven']

} ;

In this array, we can’t have a ‘firstName’. Instead, we simply have a position 0 for ‘Jonas’. In order to give our information meaning, we can use objects. In this example, we have an object with 5 key value pairs.

The firstName (etc.) is the property. The data is the value. The way we have written this object Is known as the object literal syntax.

For arrays, the order in which the data is implemented is important. For objects, it is not.

Arrays should be used for more ordered data. Objects for more unstructured data, or data that we want to name; eg first name.

**Getting Data from an Objects**

Here is our object;

*const* jonas = {

firstName: 'Jonas',

lastName: 'Schmedtmann',

age: 2037-1991,

job: 'Teacher',

friends: ['Michael', 'Peter', 'Steven']

} ;

If we want to retrieve information from our object; for example the last name. We use a ‘.’.

console.log(jonas.lastName);

Like this. We use the ‘.’ Operator. It would print Schmedtmann to the console.

We could also use bracket notation.

console.log(jonas['lastName']);

Notice that in the bracket notation we could use any expression that we like. We could compute it from an operation. An operation is an expression. We could implement that inside the brackets to output a value.

Imagine we wanted to print the full name.

*const* nameKey = 'Name';

console.log(jonas['first' + nameKey]);

console.log(jonas['last' + nameKey]);

Notice in our console.log that we have created an expression. Because nameKey is = to the string ‘Name’. We are effectively printing ‘firstName’ to the console.

This demonstrates that we can use an expression to produce values.

We can not use expressions with the ‘.’ notation. This is why bracket notation is useful to use.

Here is another example of how we can use expressions;

*const* interestedIn = prompt('What do you want to know about Jonas? Choose between firstName, lastName, age, job and friends.');

console.log(jonas[interestedIn]);

We have created a variable; interestedIn. It prompts the user for a value; firstName, lastName, age, job and friends.

We can then print the result to the console. We can not use the ‘.’ Notation. This is because we can’t use an expression. Instead we use the bracket notation. This way we can use the expression to produce a value.

In our console.log we have selected the object; Jonas. We have then implemented interestedIn, which will take the users input and compare it to the related values in Jonas’s object.

We could write some simple logic to print to the console when the user has typed something valid or invalid.

if(jonas[interestedIn]) {

console.log(jonas[interestedIn]);

} else {

console.log('Wrong request! Choose between firstName, lastName, age, job and friends.') ;

}

Here we can compare the ‘interestedIn’ value between the values stored within the Jonas object. It will return either a truthy or falsy value. If the value is truthy it will return the information requested. If the value does not match up with the data stored within the object it will return a ‘falsy’ value or ‘undefined’. This means we can use a simple if/else statement to print to the console.

**Adding Values to an Object**

jonas.location = 'Portugal';

jonas['twitter'] = '@jonasschmedtman';

console.log(jonas.location);

console.log(jonas['twitter']);

We can use bracket notation or our ‘.’ operator to add to an object. This code adds both location and a twitter handle to the object ‘Jonas’.

**Mini Challenge**

How could we write a console output that reads; ‘Jonas has 3 friends. His best friend is…’ ?

console.log(`${jonas.firstName} has ${jonas.friends.length} friends. His best friend is called ${jonas.friends[0]}.`);

Here is how we do it.

${jonas.firstName}

This, simply calls the object Jonas. We then request the first name, which is a string.

Secondly we want to determine how many friends there are in the array.

${jonas.friends.length}

We call the Jonas object. We then call the ‘friends’ array within the object. We then use the ‘.length’ property to determine the amount of data points within the array. In this case it is 3.

Finally we need to determine who Jonas’ best friend is. Assuming that it is at position 0 in the friends array!

${jonas.friends[0]}

The Jonas object is called, then the friends array, then we use [ ] to select the position of data at position 0.

This is printed to the console:



**Object Methods**

Objects, like arrays can hold different types of data. Functions can be added to objects. Any function that is added to an object is called a ‘method’.

Here is an example:

*const* jonas = {

firstName: 'Jonas',

lastName: 'Schmedtmann',

birthYear: 1991,

job: 'Teacher',

friends: ['Michael', 'Peter', 'Steven'],

hasDriversLicense: true

calcAge: *function*(*birthYear*) {

return 2037 - *birthYear*;

}

} ;

Here we have a calcAge function. In it, we pass in the birthyear as a parameter. Remember that a function is an expression that will produce a value. This is why we can use functions within objects.

Note; you can *only* use function expressions and not function declarations.

Job holds a string value. Friends holds an array value. hasDriversLicense holds a Boolean value and calcAge holds a function value.

console.log(jonas.calcAge(1991));

This is how we would call our function. We first call the object, then the function and then put in our input parameters.

We could also use [ ] bracket notation;

console.log(jonas['calcAge'](1991));

This isn’t the optimal way of writing a function within our object. This is because we already have a value for Jonas’ birth year. It would be better to call from this value directly within our function. In order to do this, we use ‘this’.

calcAge: *function* () {

console.log(this);

return 2037 - this.birthYear;

}

console.log(jonas.calcAge());

console.log(jonas['calcAge']());

This way we can call the birthyear from that property.

There is a better way of writing this code still. Here we can create a property for age with a function. We then call the property and not the function.

calcAge: *function*() {

this.age = 2037 - this.birthYear;

return this.age;

}

} ;

console.log(jonas.age());

console.log(jonas['age']());

Notice we are simply calling our age and not the calcAge function?

This is better because we only needed to calculate the age once. Then we retrieved the property with the console.log command. If we have much larger data objects/arrays, it would take time to continually calculate the same equations over and over again. By writing a function and executing it once, it saves on processing power.

Mini Challenge

How could we write; “Jonas is a 46-year old teacher. He has a/no drivers license”, by calling our code?

*const* jonas = {

firstName: 'Jonas',

lastName: 'Schmedtmann',

birthYear: 1991,

job: 'teacher',

friends: ['Michael', 'Peter', 'Steven'],

hasDriversLicense: true,

calcAge: *function*() {

this.age = 2037 - this.birthYear;

return this.age;

},

} ;

console.log(jonas.calcAge());

console.log(jonas.age);

console.log(`${jonas.firstName} is a ${jonas.age} year old, ${jonas.job}. He has ${jonas.hasDriversLicense ? 'a' : 'no'} drivers license.`);

In this method we have used a console.log to print our string with the relevant data points. But we could also put a function into our object, which we would then use to call and print this information out!

Here’s how:

*const* jonas = {

*const* jonas = {

firstName: 'Jonas',

lastName: 'Schmedtmann',

birthYear: 1991,

job: 'teacher',

friends: ['Michael', 'Peter', 'Steven'],

hasDriversLicense: true,

calcAge: *function*() {

this.age = 2037 - this.birthYear;

return this.age;

},

getSummary: *function*() {

return `${this.firstName} is a ${this.calcAge()} year old, ${this.job}. He has ${this.hasDriversLicense ? 'a' : 'no'} drivers license.`

}

} ;

console.log(jonas.getSummary());

Notice that this time we can just print to console our getSummary function. It’s much easier this way and it avoids us repeating code over and over again.

**Loops**

Loops allow us to automate repetitive tasks.

**For Loop**

While a condition is a truthy value, a **for loop keeps running**. In the line of code below the condition is when rep is less than 3; rep <= 3.

for (*let* rep = 1; rep <= 3; rep++) {

console.log(`Print this statement multiple times. Repetition number: ${rep}.`) ;

} ;

The starting point:

*let* rep = 1

This means we start at repetition number 1. Say we were repeating something 100 times, we might want to start at position 1. If we wanted a different position, we simply change this number.

The number of repetitions

rep <= 20

This command should limit the number of repetitions equal to or less than 20. This loop would continue running up until the 20th repetition.

Incremental Increase:

rep++

This command simply increases the repetition by 1 each time.

Displaying the number of repetitions

${rep}

**Looping Arrays, Breaking and Continuing**

Perhaps one of the most used aspects of for loops is to loop through arrays.

The Array:

*const* jonas = [

'Jonas', //Position 0

'Schmedtmann', //Position 1

2037 - 1991, //Position 2

'teacher', // Position 3

['Michael', 'Peter', 'Steven'] //Position 4

] ;

What if we wanted to print each array element to the console? We can use a for loop.

Remember arrays are 0 based. This means that position ‘1’ is actually position 0.

**The starting point:**

*let* i = 0

**The condition:**

i < 5

There are only 4 positions in our array. Therefore we only want a maximum of 4 to be printed to the console.

But what if we had a very long array, or we weren’t sure how many data points are in the array? We can use an expression that tells us the length of the array;

We could use this command:

jonas.length

**Increments**

i++

Increments of 1.

**The Final Code Block**

for (*let* i = 0; i < jonas.length ; i++) {

console.log(jonas[i], typeof jonas[i]);

};

**Graphical user interface, text, application

Description automatically generated**

**Filling types array**

How to create a new array based upon the values of an original array.

*const* types = [

];

for (*let* i = 0; i < jonas.length ; i++) {

console.log(jonas[i], typeof jonas[i]);

types[i] = typeof jonas[i];

};



Or we could use:

types.push(typeof jonas[i]);

**Ages**

*const* years = [1991, 2007, 1969, 2020];

*const* ages = [];

for (*let* i = 0; i < years.length; i++ ) {

ages.push(2023 - years[i]);

};

console.log(ages);

We have an array; years, that stores different year values.

We have a blank array called ‘ages’. We want to store our calculations into this array.

We use the for command. Starting at the first position (0), so i = 0.

We only want our command to run for the length or the array; years.length.

We also only want to work in increments of 1, hence; i++

In our code block we want to add our calculation to the ages array. Hence we use the ages.**push** command.

We then implement our calculation within () brackets. This calculation simply calculates how old someone is at position [i].

When logged to the console we get the list of ages.



**Looping Backwards and Loops in Loops**

Lets start with the Jonas array again;

*const* jonas = [

'Jonas',

'Schmedtmann',

2037-1991,

'Teacher',

['Michael', 'Peter', 'Steven']

];

What if we wanted a loop that worked backwards?

for (*let* i = jonas.length -1; i >= 0; i--);

Here we have: a starting point, a condition and an instruction;

**The starting point:**

i = jonas.length -1

First of all we need a start position. Remember that an array is 0 based. This means that even though there are 5 positions, the ‘4th’ position is the final position. Remember that position 1 is 0.

The length of the array is 5. Minus 1 from this and we get 4. Therefore our starting point is at position 4.

**The condition:**

i >= 0

This is our condition, that I must be greater than or equal to 0. After we have reached position 0 the loop will cease to continue as it would be producing falsy values.

**The instruction:**

i--

Finally, we can tell our loop to operate in decrements of 1. This means that it is going to move down 1 position each time. Starting with position 4 it will then go to position 3.

If we log this to the console:

for (*let* i = jonas.length -1; i >= 0; i--) {

console.log(jonas[i]);

};

Graphical user interface, text, application

Description automatically generated

We get our initial array, but in reverse.

This means that the code works.

If we use I, we get;

for (*let* i = jonas.length -1; i >= 0; i--) {

console.log(i, jonas[i]);

};

Graphical user interface, application

Description automatically generated with medium confidenceNotice that it also displays the position of each element within our array.

**Loops in Loops**

Imagine we are at the gym and starting an exercise. For the bench press, we may have 3 sets, with 10 repetitions each time. We can use iteration and loops within loops to achieve this in JavaScript.

for (*let* exercise = 1; exercise <= 3; exercise++) {

console.log(`-------Starting Exercise ${exercise}!---------`);

}

Here is an iterative loop. At the moment it only displays;

Graphical user interface, text, application

Description automatically generatedIt clearly does not display any of the reps within each exercise.

Here is how we do it:

for (*let* exercise = 1; exercise <= 3; exercise++) {

console.log(`-------Starting Exercise ${exercise}!---------`);

for (*let* reps = 10; reps >= 0; reps--){

console.log(`Repetition number; ${reps}.`)

};

} ;

For our repetitions we have chosen the starting point of 10. We then need a condition so that our code stops. We have put our condition as 0. We are going to count down from 10 to 0 and stop there. We also then need to tell the loop to move in decrements of -1.

Table

Description automatically generated

**The While Loop**

The While loop will run while a condition is true;

while (rep <= 10) {

};

In this instance, as long as the rep number is less than or equal to 10, it will continue running.

What if we wanted to create a random dice game?

*let* dice = Math.trunc(Math.random() \* 6);

console.log(dice);

First we need a random number between 1 and 6. We use Math.random() which will create a random number between 0 and 1. We can then multiply that number by 6. Here is how this works:

*let* dice = Math.random() \* 6;

console.log(`The random number is ${dice/6}, this number multipled by 6 is equal to ${dice}.`) ;



We don’t want a decimal point, so we can use the Math.trunc() method, which returns the integer part of a number by removing any fractional digits,

*let* dice = Math.trunc(Math.random() \* 6) + 1;

The Math.trunc() method.

console.log(Math.trunc(13.37));

// Expected output: 13

console.log(Math.trunc(42.84));

// Expected output: 42

console.log(Math.trunc(0.123));

// Expected output: 0

console.log(Math.trunc(-0.123));

// Expected output: -0

Look at these values, you can see that Math.trunc() is returning integers. It does NOT round up or down. This means that we could receive the number 0, but we will never receive the number 6.

This is because; 0.9 \* 6 = 5.4.

Also, 0.1 \* 6 = 0.1.

In these cases, the integer will be 5 and 0. In order to overcome this, we simply need to +1 to our formula.

What if we want to keep running the loop until we roll a 6?

*let* dice = Math.trunc(Math.random() \* 6) + 1;

while (dice !== 6) {

console.log(`You rolled a ${dice}.`) ;

dice = Math.trunc(Math.random() \* 6) + 1;

} ;

Here our while loop will continue as long as the random number generated is not 6; !==.

It is important to enter our final ‘dice = math.Trunc…..’ otherwise we would end up creating an infinite loop and the browser would crash.

Here are some results:

Roll 1:



Roll 2:

Table

Description automatically generated

Roll 3:

Table

Description automatically generated

while (dice !== 6) {

console.log(`You rolled a ${dice}.`) ;

dice = Math.trunc(Math.random() \* 6) + 1;

if (dice === 6) console.log(`Loop is about to end.`);

} ;

Table

Description automatically generated

**Continue and Break Statements**

**Continue**

Used to continue the iteration of a loop.

for (*let* i = 0; i < jonas.length ; i++) {

if(typeof jonas[i] !== 'string') continue;

console.log(jonas[i], typeof jonas[i])

} ;

**A picture containing table

Description automatically generated**

This only prints strings to the console. Hence we no longer have the age printed as it is a number. We have used an ‘is not’; !== . After each iteration we are looking for a value that is not a string. Any value other than a string will be omitted from printing to the console.

**Break**

Used to terminate the whole loop.

for (*let* i = 0; i < jonas.length ; i++) {

if(typeof jonas[i] === 'number') break;

console.log(jonas[i], typeof jonas[i])

} ;

After a number is found, the whole loop will be cancelled. Hence we only print two values to the console.

//Array

*const* jonas = [

'Jonas', //Position 0

'Schmedtmann', //Position 1

2037 - 1991, //Position 2

'teacher', // Position 3

['Michael', 'Peter', 'Steven'] //Position 4

] ;

**Looping Backwards and Loops in Loops**

Perhaps we don’t want to start at the beginning, maybe we want to start at the end and work backwards?

**The final position of the array:**

*let* i = jonas.length - 1;

We can use this simple command to start at the final position of an array. Remember there are only 4 positions within this array. So we can simply say that the length of the array (which is 5!) – 1 is equal to the final position in the array.

i >= 0

We then want to end when the final position is greater than or equal to 0.

i--

This is the decrement command. It decreases the value by 1 each time.

**Final Code Block**

for (*let* i = jonas.length - 1; i >= 0; i-- ){

console.log(i, jonas[i]);

} ;

Graphical user interface, text, application

Description automatically generated

**Loop within a Loop**

Imagine we are performing an exercise routine. We may perform 5 different exercises, however we may have 3 sets per exercise.

for (*let* exercise = 1; exercise <= 3; exercise++){

console.log(`Starting Exercise ${exercise}.`)

};

Graphical user interface, text, application

Description automatically generatedThis code would produce our initial loop.

To populate with repetitions:

for (*let* exercise = 1; exercise <= 3; exercise++){

console.log(`Starting Exercise ${exercise}.`) ;

for (*let* rep = 1; rep < 6; rep++) {

console.log(`Lifting weight rep number: ${rep}.`) ;

}

};

Table

Description automatically generatedProduces this!

**Scope**

Tightly scoping variables improves code in several ways;

1. It will make code more legible, since the blocks will organise your code into discrete sections.
2. It makes the code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
3. It’s easier to maintain your code, since your code will be modular.
4. It will save memory in your code because it will cease to exist after the block finishes running.

const logSkyColor = () => {  
  const dusk = true;  
  let color = 'blue';   
  if (dusk) {  
    let color = 'pink';  
    console.log(color); // Prints "pink"  
  }  
  console.log(color); // Prints "blue"  
};  
  
console.log(color); // throws a ReferenceError

Here you will notice;

1. Variable ‘dusk’ created within the logskyColor function.
2. After the if statement, we define a new code block with the {} braces. Here we assign a new variable to the ‘colour’ if the ‘if’ statement is truthy.
3. Within the if block, the colour variable holds the value ‘pink’, though outside the if block, in the function body, the colour variable holds the value ‘blue’.
4. While we use block scope we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.