Synchronous vs Asynchronous

Normally, programming languages are synchronous, and some provide a way to manage asynchronicity, in the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, Python, they are all synchronous by default. Some of them handle async by using threads, spawning a new process.

JavaScript is synchronous by default and is single threaded. This means that code cannot create new threads and run in parallel.

Lines of code are executed in series, one after another, for example:

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

But JavaScript was born inside the browser, its main job, in the beginning, was to respond to user actions, like onClick, onMouseOver, onChange, onSubmit and so on. How could it do this with a synchronous programming model?

The answer was in its environment. The browser provides a way to do it by providing a set of APIs that can handle this kind of functionality.

More recently, Node.js introduced a non-blocking I/O environment to extend this concept to file access, network calls and so on.

You can’t know when a user is going to click a button, so what you do is, you define an event handler for the click event. This event handler accepts a function, which will be called when the event is triggered:

```
document.getElementById('button').addEventListener('click', () => {
 //item clicked
})
```

This is the so-called **callback**.

It’s common to wrap all your client code in a load event listener on the windowobject, which runs the callback function only when the page is ready:

```
window.addEventListener('load', () => {
 //window loaded
 //do what you want
})
```

Callbacks are used everywhere, not just in DOM events.

One common example is by using timers:

```
setTimeout(() => {
 // runs after 2 seconds
}, 2000)
```

The problem with callbacks

Callbacks are great for simple cases!

However every callback adds a level of nesting, and when you have lots of callbacks, the code starts to be complicated very quickly:

```
window.addEventListener('load', () => {
 document.getElementById('button').addEventListener('click', () => {
   setTimeout(() => {
     items.forEach(item => {
       //your code here
     })
   }, 2000)
 })
})
```

This is just a simple 4-levels code, but I’ve seen much more levels of nesting and it’s not fun.

How do we solve this?

ALTERNATIVES TO CALLBACKS

Starting with ES6, JavaScript introduced several features that help us with asynchronous code that do not involve using callbacks:

- Promises (ES6)
- Async/Await (ES8)

Synchronous:

1. If an API call is synchronous, it means that code execution will block (or wait) for the API call to return before continuing. This means that until a response is returned by the API, your application will not execute any further, which could be perceived by the user as latency or performance lag in your app. Making an API call synchronously can be beneficial, however, if there if code in your app that will only execute properly once the API response is received.

Asynchronous:

1. Asynchronous calls do not block (or wait) for the API call to return from the server. Execution continues on in your program, and when the call returns from the server, a "callback" function is executed.

Websocket:

https://docs.google.com/document/d/1k8o32uCivNQAnvpVSOGgboK77oGMKNGIdXu7bzx3vpE/edit

Java script:

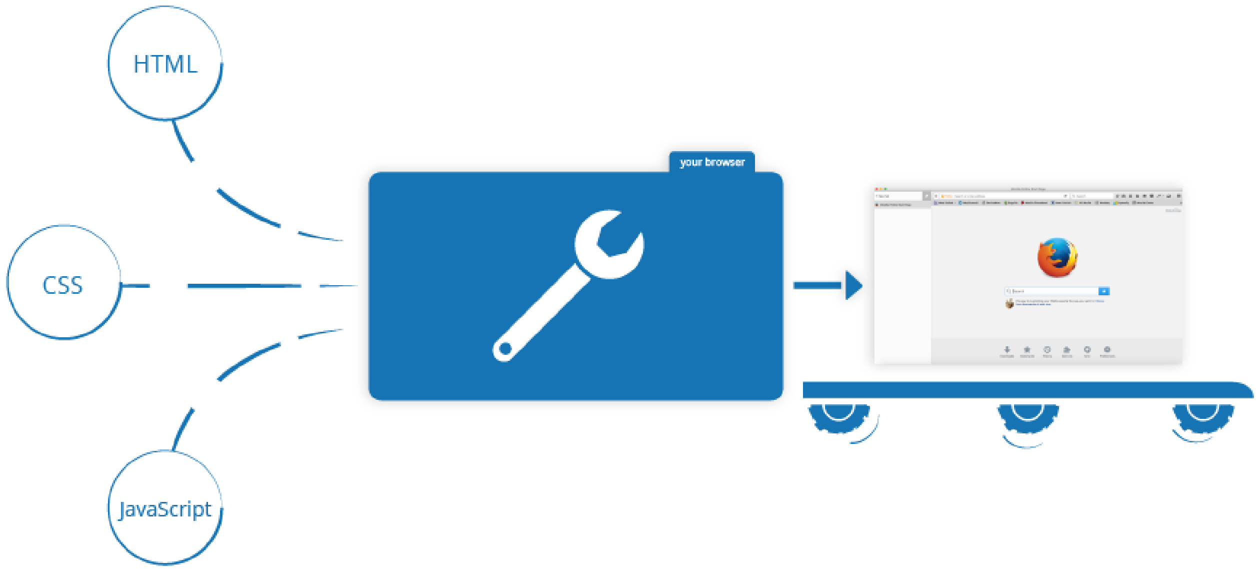
JavaScript® (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well.

JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behaviour.

JavaScript is an object-based language **based on prototypes**, rather than being class-based.

JavaScript is a programming language that allows you to implement complex things on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, or interactive maps, or animated 2D/3D graphics, or scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved.

What happens behind the browser:



The JavaScript is executed by the **browser's JavaScript engine**, after the HTML and CSS have been assembled and put together into a web page. This ensures that the structure and style of the page are already in place by the time the JavaScript starts to run.

This is a good thing, as a very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (as mentioned above). If the JavaScript loaded and tried to run before the HTML and CSS was there to affect, then errors would occur.

Browser security:

Each browser tab is its own separate bucket for running code in (these buckets are called "execution environments" in technical terms) — this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab — or on another website. This is a good security measure — if this were not the case, then pirates could start writing code to steal information from other websites, and other such bad things.

What are browser developer tools:

Every modern web browser includes a powerful suite of developer tools. These tools do a range of things, from inspecting currently-loaded HTML, CSS and JavaScript to showing which assets the page has requested and how long they took to load.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

Code:

```
var para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {
  var name = prompt('Enter a new name');
  para.textContent = 'Player 1: ' + name;
}
```

Here we are selecting a text paragraph (line 1), then attaching an event listener to it (line 3) so that when the paragraph is clicked, the updateName() code block (lines 5–8) is run.

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the browser developer console — TypeError: para is undefined. This means that the para object does not exist yet, so we can't add an event listener to it.

Interpreted versus compiled code:

You might hear the terms interpreted and compiled in the context of programming. JavaScript is an **interpreted language** — the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example C/C++ are compiled into assembly language that is then run by the computer.

How do you add java script into your page:

Internal:

```
<script>  </script>
```

External:

```
<script src=”script.js”> </script>
```

Inline java script handlers:

```
function createParagraph() {
  var para = document.createElement('p');
  para.textContent = 'You clicked the button!';
  document.body.appendChild(para);
}
```

<button onclick="createParagraph()">Click me!</button>

Note : Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the onclick="createParagraph()" attribute on every button you wanted the JavaScript to apply to.

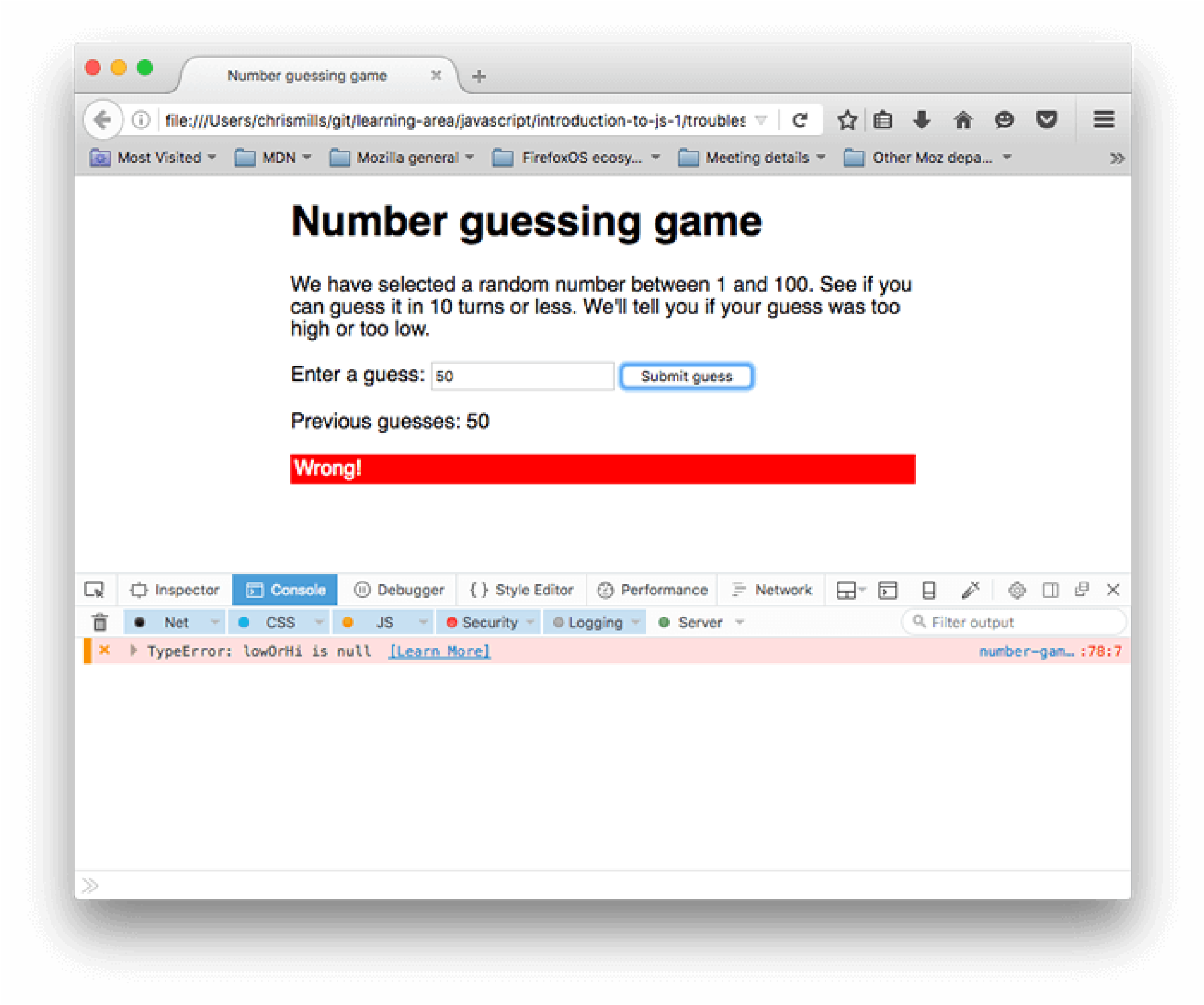
Comment:

Single Line : // , Multi - line comment : /* */

Types of error:

- **Syntax errors**:
    - These are spelling errors in your code that actually cause the program not to run at all, or stop working part way through — you will usually be provided with some error messages too. These are usually okay to fix, as long as you are familiar with the right tools and know what the error messages mean!
- **Logic errors:**
    - These are errors where the syntax is actually correct but the code is not what you intended it to be, meaning that program runs successfully but gives incorrect results. These are often harder to fix than syntax errors, as there usually isn't a resulting error message to direct you to the source of the error.

Fixing syntax error:



Number guessing game

We have selected a random number between 1 and 100. See if you can guess it in 10 turns or less. We'll tell you if your guess was too high or too low.

Enter a guess: 50

Previous guesses: 50

Wrong!

```
TypeError: lowOrHi is null  [Learn More]                number-gam… :78:7
```

Here,

- A red "x" to indicate that this is an error.
- An error message to indicate what's gone wrong: "TypeError: guessSubmit.addeventListener is not a function"
- A "Learn More" link that links through to an MDN page that explains what this error means in huge amounts of detail.

- The name of the JavaScript file, which links through to the Debugger tab of the devtools. If you follow this link, you'll see the exact line where the error is highlighted.
- The line number where the error is, and the character number in that line where the error is first seen. In this case, we've got line 86, character number 3.

Note : console.log() is a really useful debugging function that prints a value to the console.

Java script error ref:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors

Java script variables:

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence. But one special thing about variables is that their contained values can change. Let's look at a simple example:

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are Number_hits, temp99, $credit, and _name.

Evaluating variables:

A variable declared using the var or let statement with no assigned value specified has the value of undefined.

An attempt to access an undeclared variable results in a ReferenceError exception being thrown:

```
var a;
console.log('The value of a is ' + a); // The value of a is undefined
console.log('The value of b is ' + b); // The value of b is undefined
var b;
console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not defined
let x;
console.log('The value of x is ' + x); // The value of x is undefined
console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined
let y;
```

You can use undefined to determine whether a variable has a value. In the following code, the variable input is not assigned a value, and the if statement evaluates to true.

```
var input;
if (typeof input === undefined) {
  doThis();
} else {
  doThat();
}
```

Code:

```
<button>Press me</button>
```

```
var button = document.querySelector('button');
button.onclick = function() {
  var name = prompt('What is your name?');
  alert('Hello ' + name + ', nice to see you!');
}
```

Note : If you write a multiline JavaScript program that declares and initializes a variable, you can actually declare it after you initialize it and it will still work. This is because variable declarations are generally done first before the rest of the code is executed. This is called **hoisting** — read var hoisting for more detail on the subject.

- Variables are case sensitive — so myage is a different variable to myAge.
- A safe convention to stick to is so-called "lower camel case", where you stick together multiple words, using lower case for the whole first word and then capitalize subsequent words. We've been using this for our variable names in the article so far.
- Avoid using JavaScript reserved words as your variable names — by this, we mean the words that make up the actual syntax of JavaScript! So you can't use words like var, function, let, and for as variable names. Browsers will recognize them as different code items, and so you'll get errors.

Note : You can find a fairly complete list of reserved keywords to avoid at Lexical grammar — keywords.

Scope of a variable:

Assigning a value to an undeclared variable implicitly creates it as a global variable (it becomes a property of the global object) when the assignment is executed. The differences between declared and undeclared variables are:

1. Declared variables are constrained in the execution context in which they are declared. Undeclared variables are always global.

```
function x() {
 y = 1;   // Throws a ReferenceError in strict mode
 var z = 2;
}
```

```
x();

console.log(y); // logs "1"
console.log(z); // Throws a ReferenceError: z is not defined outside x
```

3. Declared variables are a non-configurable property of their execution context (function or global). Undeclared variables are configurable (e.g. can be deleted).

```
var a = 1;
b = 2;

delete this.a; // Throws a TypeError in strict mode. Fails silently otherwise.
delete this.b;

console.log(a, b); // Throws a ReferenceError.
```

```
4. Var x= 10 ;
Var y = 10;
{
Var x= 6;
Let y=6;
}
console.log(x) // prints 6
console.log(y) // prints 10
```

Var hoisting:

Because variable declarations (and declarations in general) are processed before any code is executed, declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared.

This behavior is called "hoisting", as it appears that the variable declaration is moved to the top of the function or global code.

```
bla = 2;
var bla;
// ...

// is implicitly understood as:

var bla;
bla = 2;
```

For that reason, it is recommended to always declare variables at the top of their scope (the top of global code and the top of function code) so it's clear which variables are function scoped (local) and which are resolved on the scope chain.

It's important to point out that the hoisting will affect the variable declaration, but not its value's initialization. The value will be indeed assigned when the assignment statement is reached:

```
function do_something() {
  console.log(bar); // undefined
  var bar = 111;
  console.log(bar); // 111
}

// is implicitly understood as:
function do_something() {
  var bar;
  console.log(bar); // undefined
  bar = 111;
  console.log(bar); // 111
}
```

Variable types:

Numbers

You can store numbers in variables, either whole numbers like 30 (also called integers) or decimal numbers like 2.456 (also called floats or floating point numbers). You don't need to declare variable types in JavaScript, unlike some other programming languages. When you give a variable a number value, you don't include quotes:

```
var myAge = 17;
```

Booleans

Booleans are true/false values — they can have two values, true or false. These are generally used to test a condition, after which code is run as appropriate. So for example, a simple case would be:

```
var iAmAlive = true;
```

Whereas in reality it would be used more like this:

```
var test = 6 < 3;
```

This is using the "less than" operator (<) to test whether 6 is less than 3. As you might expect, it will return false, because 6 is not less than 3! You will learn a lot more about such operators later on in the course.

Data type conversion

JavaScript is a **dynamically typed language**. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42;
```

And later, you could assign the same variable a string value, for example:

```
answer = 'Thanks for all the fish...';
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = 'The answer is ' + 42 // "The answer is 42"
y = 42 + ' is the answer' // "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
'37' - 7 // 30
'37' + 7 // "377"
```

Exception handling statements:

You can throw exceptions using the throw statement and handle them using the try...catch statements.

Throw statement:

Use the throw statement to throw an exception. When you throw an exception, you specify the expression containing the value to be thrown:

```
throw expression;
```

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

```
throw 'Error2';   // String type
throw 42;         // Number type
throw true;       // Boolean type
throw {toString: function() { return "I'm an object!"; } };
```

Note: You can specify an object when you throw an exception. You can then reference the object's properties in the catch8080 block.

try…. catch statement

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block. That is, you want the try block to succeed, and if it does not succeed, you want control to pass to the catch block.

The following example uses a try...catch statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1-12), an exception is thrown with the value "InvalidMonthNo" and the statements in the catch block set the monthNamevariable to unknown.

```
function getMonthName(mo) {
  mo = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)
  var months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
  if (months[mo]) {
    return months[mo];
  } else {
    throw 'InvalidMonthNo'; //throw keyword is used here
  }
}

try { // statements to try
  monthName = getMonthName(myMonth); // function could throw exception
}
catch (e) {
  monthName = 'unknown';
  logMyErrors(e); // pass exception object to error handler -> your own function
}
```

The finally block

The finally block contains statements to execute after the try and catch blocks execute but before the statements following the try...catch statement. The finally block executes whether or not an exception is thrown. If an exception is thrown, the statements in the finally block execute even if no catch block handles the exception.

```
openMyFile();
try {
  writeMyFile(theData); //This may throw an error
} catch(e) {
  handleError(e); // If we got an error we handle it
} finally {
  closeMyFile(); // always close the resource
}
```

If the finally block returns a value, this value becomes the return value of the entire try-catch-finally production, regardless of any return statements in the try and catch blocks:

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch(e) {
    console.log(1);
    return true; // this return statement is suspended
                 // until finally block has completed
    console.log(2); // not reachable
  } finally {
    console.log(3);
    return false; // overwrites the previous "return"
    console.log(4); // not reachable
  }
  // "return false" is executed now
  console.log(5); // not reachable
}
f(); // console 0, 1, 3; returns false
```

Overwriting of return values by the finally block also applies to exceptions thrown or re-thrown inside of the catch block:

```
function f() {
  try {
    throw 'bogus';
  } catch(e) {
    console.log('caught inner "bogus"');
    throw e; // this throw statement is suspended until
             // finally block has completed
  } finally {
    return false; // overwrites the previous "throw"
  }
  // "return false" is executed now
}

try {
  f();
} catch(e) {
  // this is never reached because the throw inside
  // the catch is overwritten
  // by the return in finally
  console.log('caught outer "bogus"');
}

// OUTPUT
// caught inner "bogus"
```

Numbers:

There is no specific type for integers. In addition to being able to represent floating-point numbers, the number type has three symbolic values: +Infinity, -Infinity, and NaN (not-a-number). See also JavaScript data types and structures for context with other primitive types in JavaScript.

You can use four types of number literals: decimal, binary, octal, and hexadecimal.

Decimal numbers

```
1234567890
42
// Caution when using leading zeros:
0888 // 888 parsed as decimal
0777 // parsed as octal in non-strict mode (511 in decimal)
```

Note that decimal literals can start with a zero (0) followed by another decimal digit, but if every digit after the leading 0 is smaller than 8, the number gets parsed as an octal number.

Binary:
Binary number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "B" (0b or 0B). If the digits after the 0b are not 0 or 1, the following SyntaxError is thrown: "Missing binary digits after 0b".

```
var FLT_SIGNBIT  = 0b10000000000000000000000000000000; // 2147483648
var FLT_EXPONENT = 0b01111111100000000000000000000000; // 2139095040
var FLT_MANTISSA = 0B00000000011111111111111111111111; // 8388607
```

Octal numbers:

Octal number syntax uses a leading zero. If the digits after the 0 are outside the range 0 through 7, the number will be interpreted as a decimal number.

```
var n = 0755; // 493
var m = 0644; // 420
```

In ECMAScript 2015, octal numbers are supported if they are prefixed with `0o`, e.g.:

```
var a = 0o10; // ES2015: 8
```

Hexadecimal numbers:

Hexadecimal number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "X" (0x or 0X). If the digits after 0x are outside the range (0123456789ABCDEF), the following SyntaxError is thrown: "Identifier starts immediately after numeric literal".

```
0xFFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF   // 81985529216486900
0XA                 // 10
```

Exponentiation

```
1E3   // 1000
2e6   // 2000000
0.1e2 // 10
```

Number object:

The built-in Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

Methods of Number:

```
Number.parseFloat(3.5444) // returns true
Number.parseInt(3) // returns true
Number.isInteger(5) // returns true
Number.isInteger(5.2) // returns false
Number.isNaN()
```

Math Object:

The built-in Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which you would use in an application as

```
console.log(Math.PI)
```

For Math methods ,
https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Numbers_and_dates

Date object:

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

```
var Xmas95 = new Date('December 25, 1995');
```

For more ,
https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Numbers_and_dates

----

Spread operator (...)

Syntax

```
For function calls -- > myFunction(...iterableObj);
For array literals or string -- > [...iterableObj, '4', 'five', 6];
For object literals -- > let objClone = { ...obj };
```

In function call

```
function sum(x, y, z) {
  return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // expected output: 6
```

In array literals

Without spread syntax, to create a new array using an existing array as one part of it, the array literal syntax is no longer sufficient and imperative code must be used instead using a combination of push, splice, concat, etc. With spread syntax this becomes much more succinct:

```
var parts = ['shoulders', 'knees'];
var lyrics = ['head', ...parts, 'and', 'toes']; // ["head", "shoulders", "knees", "and", "toes"]
```

Copy an array

```
var arr = [1, 2, 3];
var arr2 = [...arr]; // like arr.slice()
```

A better way to concatenate arrays

Array.concat is often used to concatenate an array to the end of an existing array. Without spread syntax this is done as:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = arr1.concat(arr2); // Append all items from arr2 onto arr1
```

With spread syntax this becomes:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = [...arr1, ...arr2]; // arr1 is now [0, 1, 2, 3, 4, 5]
```

Spread in shift & unshift opeations

Array.unshift is often used to insert an array of values at the start of an existing array. Without spread syntax this is done as:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5]; // Prepend all items from arr2 onto arr1
Array.prototype.unshift.apply(arr1, arr2) // arr1 is now [3, 4, 5, 0, 1, 2]
```

With spread syntax, this becomes [Note, however, that this creates a new arr1 array. Unlike Array.unshift, it does not modify the original arr1 array in-place]:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = [...arr2, ...arr1]; // arr1 is now [3, 4, 5, 0, 1, 2]
```

Spread in Object literals

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };
var clonedObj = { ...obj1 }; // Object { foo: "bar", x: 42 }
var mergedObj = { ...obj1, ...obj2 }; // Object { foo: "baz", x: 42, y: 13 }
```

Spread vs Rest

Rest syntax looks exactly like spread syntax but is used for destructuring arrays and objects. In a way, rest syntax is the opposite of spread syntax: spread 'expands' an array into its elements, while rest collects multiple elements and 'condenses' them into a single element.

Destructuring assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible **to unpack values from arrays, or properties from objects, into distinct variables**.

```
[a, b] = [10, 20];
console.log(a);
// expected output: 10
console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(rest);
// expected output: [30,40,50]

({ a, b } = { a: 10, b: 20 });
console.log(a); // 10
console.log(b); // 20

({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});
console.log(a); // 10
console.log(b); // 20
console.log(rest); // {c: 30, d: 40}

var x = [1, 2, 3, 4, 5];
var [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```

Strings

Strings are pieces of text. When you give a variable a string value, you need to wrap it in single or double quote marks, otherwise JavaScript will try to intepret it as another variable name.

Strings as objects

We've said it before, and we'll say it again — ***everything* is an object in JavaScript**. When you create a string, for example by using

```
var string = 'This is my string';
```

your variable becomes a string object instance, and as a result has a large number of properties and methods available to it. You can see this if you go to the String object page and look down the list on the side of the page!

Long literal strings:

Sometimes, your code will include strings which are very long. Rather than having lines that go on endlessly, or wrap at the whim of your editor, you may wish to specifically break the string into multiple lines in the source code without affecting the actual string contents. There are two ways you can do this.

You can use the + operator to append multiple strings together, like this:

```
let longString = "This is a very long string which needs " +
                 "to wrap across multiple lines because " +
                 "otherwise my code is unreadable.";
```

Or you can use the backslash character ("\") at the end of each line to indicate that the string will continue on the next line. Make sure there is no space or any other character after the backslash (except for a line break), or as an indent; otherwise it will not work. That form looks like this:

```
let longString = "This is a very long string which needs \
to wrap across multiple lines because \
otherwise my code is unreadable.";
```

Character access

There are two ways to access an individual character in a string. The first is the charAt() method:

```
return 'cat'.charAt(1); // returns "a"
```

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

```
return 'cat'[1]; // returns "a"
```

Finding the length of a string :

This is easy — you simply use the length property. Try entering the following lines:

```
var browserType = 'mozilla';
browserType.length;
```

Comparing the length of a string :

C developers have the strcmp() function for comparing strings. In JavaScript, you just use the less-than and greater-than operators:

```
var a = 'a';
var b = 'b';
if (a < b) { // true
```

```
  console.log(a + ' is less than ' + b);
} else if (a > b) {
  console.log(a + ' is greater than ' + b);
} else {
  console.log(a + ' and ' + b + ' are equal.');
}
```

Distinction between string primitives and String objects

Note that JavaScript distinguishes between String objects and primitive string values. (The same is true of Boolean and Numbers.)

String literals (denoted by double or single quotes) and strings returned from String calls in a non-constructor context (i.e., without using the new keyword) are primitive strings. **JavaScript automatically converts primitives to String objects**, so that it's possible to use String object methods for primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup.

```
var s_prim = 'foo';
var s_obj = new String(‘foo’);

console.log(s_prim) // ‘foo’
console.log(s_obj) // {'0': 'f', '1': 'o', '2': 'o'}
console.log(typeof s_prim); // Logs "string"
console.log(typeof s_obj);  // Logs "object"
```

String primitives and String objects also give different results when using eval(). Primitives passed to eval are treated as source code; String objects are treated as all other objects are, by returning the object. For example:

```
var s1 = '2 + 2';             // creates a string primitive
var s2 = new String('2 + 2'); // creates a String object
console.log(eval(s1));        // returns the number 4
console.log(eval(s2));        // returns the string "2 + 2"
```

For these reasons, code may break when it encounters String objects when it expects a primitive string instead, although generally authors need not worry about the distinction.

A String object can always be converted to its primitive counterpart with the valueOf() method.

```
console.log(eval(s2.valueOf())); // returns the number 4
```

For more string methods ,
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Valid string examples :

```
var sgl = 'Single quotes.';
var dbl = "Double quotes";

var sglDbl = 'Would you eat a "fish supper"?';
var dblSgl = "I'm feeling blue.";

var bigmouth = 'I\'ve got no right to take my place...';

var dolphinGoodbye = 'So long and thanks for all the fish';
```

String interpolation:

The JavaScript term for inserting the data saved to a variable into a string is *string interpolation*.

```
let myPet= “dog”
console.log(‘I own a pet’ + myPet + ‘.’);
```

In the newest version of JavaScript (ES6) we can insert variables into strings with ease, by using ` symbol instead of quotes and with $ symbol.

```
let myName="Gautham";

let myCity="Chennai";

console.log(`My name is ${myName}. My favorite city is ${myCity}`);
```

Using special characters in strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

\0	Null Byte
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\'	Apostrophe or single quote
\"	Double quote
\\	Backslash character

Escaping characters

For characters not listed in the table, a preceding backslash is ignored, but this usage is deprecated and should be avoided.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example:

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service.";
console.log(quote);
```

The result of this would be:

```
He read "The Cremation of Sam McGee" by R.W. Service.
```

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path c:\temp to a string, use the following:

```
var home = 'c:\\temp';
```

Multi line strings:

Any new line characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log('string text line 1\n\
string text line 2');
// "string text line 1
// string text line 2"
```

To get the same effect with multi-line strings, you can now write:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Hexadecimal escape sequences

The number after \x is interpreted as a hexadecimal number.

```
'\xA9' // "©"
```

Unicode escape sequences

The Unicode escape sequences require at least four hexadecimal digits following \u.

```
'\u00A9' // "©"
```

Note: You can't change individual characters because strings are immutable array-like objects:

```
var mystring = 'Hello, World!';
var x = mystring.length;
mystring[0] = 'L'; // This has no effect, because strings are immutable
mystring[0]; // This returns "H"
```

For more ,
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting

----

Regular expressions:

Regular expressions are **patterns used to match character combinations in strings**.

Creating a regular expression

You construct a regular expression in one of two ways:

Using a regular expression literal, which consists of a pattern enclosed between slashes, as follows:

```
var re = /ab+c/;
```

Regular expression literals provide compilation of the regular expression when the script is loaded. If the regular expression remains constant, using this can improve performance.

Or calling the constructor function of the RegExp object, as follows:

```
var re = new RegExp('ab+c');
```

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

Writing a regular expression pattern

A regular expression pattern is composed of simple characters, such as /abc/, or a combination of simple and special characters, such as /ab*c/ or /Chapter (\d+)\.\d*/.

Using simple patterns:

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern /abc/ matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string 'Grab crab' because while it contains the substring 'ab c', it does not contain the exact substring 'abc'.

Using special characters:

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, the pattern includes special characters. For example, the pattern /ab*c/ matches any character combination in which a single 'a' is followed by zero or more 'b's (* means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

For more operators,
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Working with regular expression:

Regular expressions are used with the RegExp methods test and exec and with the String methods match, replace, search, and split.

```
var paragraph = 'The quick brown fox jumped over the lazy dog. It barked.';
var regex = /[A-Z]/g;
var found = paragraph.match(regex);
console.log(found); // [ “T” ,  “I” ]
```

When you want to know whether a pattern is found in a string, use the test or search method; for more information (but slower execution) use the exec or match methods.

In the following example, the script uses the exec method to find a match in a string.

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec('cdbbdbsbz');
```

Arrays

An array is a single object that contains multiple values enclosed in square brackets and separated by commas.

Creating an array:

The following statements create equivalent arrays:

```
var arr = new Array(element0, element1, ..., elementN);
var arr = Array(element0, element1, ..., elementN);
var arr = [element0, element1, ..., elementN];
```

The bracket syntax is called an "array literal" or "array initializer." It's shorter than other forms of array creation, and so is generally preferred. See Array literals for details.

```
var myNameArray = ['Chris', 'Bob', 'Jim'];
var myNumberArray = [10,15,40];
```

Once these arrays are defined, you can access each value by their location within the array. Try these lines:

```
myNameArray[0]; // should return 'Chris'
myNumberArray[2]; // should return 40
```

To create an array with non-zero length, but without any items, either of the following can be used:

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);
I
// This has exactly the same effect
var arr = [];
arr.length = arrayLength;
```

In addition to a newly defined variable as shown above, arrays can also be assigned as a property of a new or an existing object:

```
var obj = {};
// ...
obj.prop = [element0, element1, ..., elementN];

// OR
var obj = {prop: [element0, element1, ...., elementN]};
```

If you wish to initialize an array with a single element, and the element happens to be a Number, you must use the bracket syntax. When a single Number value is passed to the Array() constructor or function, it is interpreted as an arrayLength, not as a single element.

```
var arr = [42];      // Creates an array with only one element:
                     // the number 42.

var arr = Array(42); // Creates an array with no elements
                     // and arr.length set to 42; this is
                     // equivalent to:
var arr = [];
arr.length = 42;
```

Calling Array(N) results in a RangeError, if N is a non-whole number whose fractional portion is non-zero. The following example illustrates this behavior.

```
var arr = Array(9.3);  // RangeError: Invalid array length
```

Populating an array:

```
var emp = [];
emp[0] = 'Casey Jones';
emp[1] = 'Phil Lesh';
emp[2] = 'August West';
```

You can also populate an array when you create it:

```
var myArray = new Array('Hello', myVar, 3.14159);
var myArray = ['Mango', 'Apple', 'Orange'];
```

Understanding length:

The length property is special; it always returns the index of the last element plus one (in the following example, Dusty is indexed at 30, so cats.length returns 30 + 1). Remember, JavaScript Array indexes are 0-based: they start at 0, not 1. This means that the length property will be one more than the highest index stored in the array:

```
var cats = [];
cats[30] = ['Dusty'];
console.log(cats.length); // 31 (creates empty values in between)

var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length); // 3


cats.length = 2;
console.log(cats); // logs "Dusty, Misty" - Twiggy has been removed


cats.length = 0;
console.log(cats); // logs []; the cats array is empty


cats.length = 3;
console.log(cats); // logs [ <3 empty items> ]
```

Array methods:

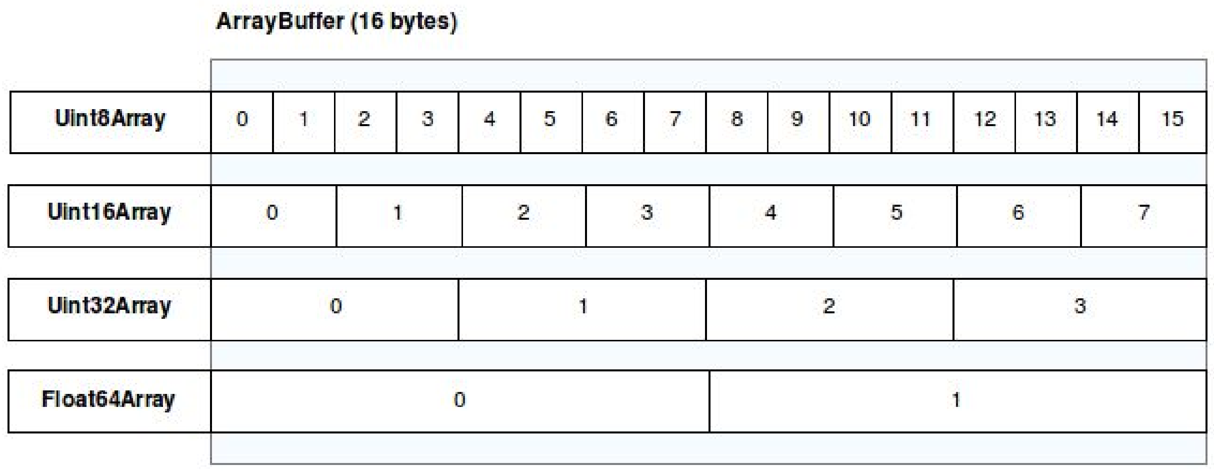
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections

Typed array:

JavaScript **typed arrays** are array-like objects and provide a **mechanism for accessing raw binary data**. As you may already know, Array objects grow and shrink dynamically and can have any JavaScript value. JavaScript engines perform optimizations so that these **arrays** are fast.

Buffers and views: typed array architecture

To achieve maximum flexibility and efficiency, JavaScript typed arrays split the implementation into buffers and views. A buffer (implemented by the ArrayBufferobject) is an object representing a chunk of data; it has no format to speak of, and offers no mechanism for accessing its contents. In order to access the memory contained in a buffer, you need to use a view. A view provides a context — that is, a data type, starting offset, and number of elements — that turns the data into an actual typed array.



Array Buffer:

The ArrayBuffer is a data type that is used to represent a generic, fixed-length binary data buffer. You can't directly manipulate the contents of an ArrayBuffer; instead, you create a typed array view or a DataView which represents the buffer in a specific format, and use that to read and write the contents of the buffer.

Typed array views:

Typed array views have self descriptive names and provide views for all the usual numeric types like Int8, Uint32, Float64 and so forth. There is one special typed array view, the Uint8ClampedArray. It clamps the values between 0 and 255. This is useful for Canvas data processing, for example.

For more,

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

----

Keyed collection:

Maps:

ECMAScript 2015 introduces a new data structure to map values to values. A Map object is a simple key/value map and can iterate its elements in insertion order.

The following code shows some basic operations with a Map. See also the Map reference page for more examples and the complete API. You can use a for...of loop to return an array of [key, value] for each iteration.

```
var sayings = new Map();
sayings.set('dog', 'woof');
sayings.set('cat', 'meow');
sayings.set('elephant', 'toot');
sayings.size; // 3
sayings.get('fox'); // undefined
sayings.has('bird'); // false
sayings.delete('dog');
sayings.has('dog'); // false

for (var [key, value] of sayings) {
  console.log(key + ' goes ' + value);
}
// "cat goes meow"
// "elephant goes toot"
```

```
sayings.clear();
sayings.size; // 0
```

Traditionally, objects have been used to map strings to values. Objects allow you to set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Map objects, however, have a few more advantages that make them better maps.

- The keys of an Object are Strings, where they can be of any value for a Map.
- You can get the size of a Map easily while you have to manually keep track of size for an Object.
- The iteration of maps is in insertion order of the elements.
- An Object has a prototype, so there are default keys in the map. (this can be bypassed using map = Object.create(null))

These three tips can help you to decide whether to use a Map or an Object:

- Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.
- Use objects when there is logic that operates on individual elements.

Sets:

Set objects are collections of values. You can iterate its elements in insertion order. A value in a Set may only occur once; it is **unique** in the Set's collection.

The following code shows some basic operations with a Set. See also the Setreference page for more examples and the complete API.

```
var mySet = new Set();
mySet.add(1);
mySet.add('some text');
mySet.add('foo');

mySet.has(1); // true
mySet.delete('foo');
mySet.size; // 2

for (let item of mySet) console.log(item);
// 1
// "some text"
```

Converting between Arrays & Sets

You can create an Array from a Set using Array.from or the spread operator. Also, the Set constructor accepts an Array to convert in the other direction. Note again that Set objects store unique values, so any duplicate elements from an Array are deleted when converting.

```
Array.from(mySet);
mySet2 = new Set([1, 2, 3, 4]);
```

Array & Set compared

Traditionally, a set of elements has been stored in arrays in JavaScript in a lot of situations. The new Set object, however, has some advantages:

- Checking whether an element exists in a collection using indexOf for arrays is slow.
- Set objects let you delete elements by their value. With an array you would have to splice based on an element's index.
- The value NaN cannot be found with indexOf in an array.
- Set objects store unique values; you don't have to keep track of duplicates by yourself.

----

this:

A function's this keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

In most cases, the value of **this is determined by how a function is called.** It can't be set by assignment during execution, and it may be different each time the function is called. **ES5 introduced the bind method** to set the value of a function's this regardless of how it's called, and **ES2015 introduced arrow functions** which don't provide their own this binding (it retains the this value of the enclosing lexical context).

```
var test = {
  prop: 42,
  func: function() {
    return this.prop;
  },
};

console.log(test.func());
// expected output: 42
```

Global context:

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true


a = 37;
console.log(window.a); // 37


this.b = "MDN";
console.log(window.b)  // "MDN"
console.log(b)         // "MDN"
```

Function context:

Inside a function, the value of this depends on how the function is called.

Simple call:

Since the following code is not in strict mode, and because the value of this is not set by the call, this will default to the global object, which is window in a browser.

```
function f1() {
  return this;
}

// In a browser:
f1() === window; // true

// In Node:
f1() === global; // true
```

In strict mode, however, the value of this remains at whatever it was set to when entering the execution context, so, in the following case, this will default to undefined**:**

```
function f2() {
  'use strict'; // see strict mode
  return this;
}

f2() === undefined; // true
```

So, in **strict mode,** if this was not defined by the execution context, it remains undefined.

bind vs call vs apply

Use `.bind()` when you want that function to later be called with a certain context, useful in events. Use `.call()` or `.apply()` when you want to call the function immediately, and modify the context.

An object can be passed as the first argument to call or apply and this will be bound to `//` it.

```
var obj = {a: 'Custom'};

// This property is set on the global object
var a = 'Global';

function whatsThis() {
  return this.a;  // The value of this is dependent on how the function is called
}

whatsThis();          // 'Global'
```

```
whatsThis.call(obj);  // 'Custom'
'whatsThis.apply(obj); // 'Custom
```

Ex: 2

```
function add(c, d) {
  return this.a + this.b + c + d;
}

var o = {a: 1, b: 3};

// The first parameter is the object to use as
// 'this', subsequent parameters are passed as
// arguments in the function call
add.call(o, 5, 7); // 16

// The first parameter is the object to use as
// 'this', the second is an array whose
// members are used as the arguments in the function call
add.apply(o, [10, 20]); // 34
```

The bind method:

ECMAScript 5 introduced Function.prototype.bind. Calling f.bind(someObject)creates a new function with the same body and scope as f, but where this occurs in the original function, in the new function it is permanently bound to the first argument of bind, regardless of how the function is being used.

```
function f() {
  return this.a;
}
```

```
var g = f.bind({a: 'azerty'});
console.log(g()); // azerty

var h = g.bind({a: 'yoo'}); // bind only works once!
console.log(h()); // azerty

var o = {a: 37, f: f, g: g, h: h};
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

Arrow function:

In arrow functions, this retains the value of the enclosing lexical context's this. In global code, it will be set to the global object:

```
var globalObject = this;
var foo = () => this; //global function
console.log(foo() === globalObject); // true
```

As an object method:

When a function is called as a method of an object, its this is set to the object the method is called on.

In the following example, when o.f() is invoked, inside the function this is bound to the o object.

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // 37
```

Note that this behavior is not at all affected by how or where the function was defined. In the previous example, we defined the function inline as the f member during the definition of o. However, we could have just as easily defined the function first and later attached it to o.f. Doing so results in the same behavior:

```
var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // 37

o.b = {g: independent, prop: 42};
console.log(o.b.g()); // 42
```

this on the object's prototype chain

The same notion holds true for methods defined somewhere on the object's prototype chain. If the method is on an object's prototype chain, this refers to the object the method was called on, as if the method were on the object.

```
var o = {f: function() { return this.a + this.b; }};
var p = Object.create(o);
p.a = 1;
p.b = 4;
```

```
console.log(p.f()); // 5
```

As a constructor:

When a function is used as a constructor (with the new keyword), its this is bound to the new object being constructed.

```
function C() {
  this.a = 37;
}

var o = new C();
console.log(o.a); // 37

function C2() {
  this.a = 37;
  return {a: 38};
}
o = new C2();
console.log(o.a); // 38
```

Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

----

Objects

In programming, an object is a structure of code that models a real life object. You can have a simple object that represents a car park and contains information about its width and length, or you could have an object that represents a person, and contains data about their name, height, weight, what language they speak, how to say hello to them, and more.

Try entering the following line into your console:

```
var person = { name : 'Trump’, age: 70 };
```

To retrieve the information stored in the object, you can use the following syntax:

```
person.name
```

JavaScript is a "**loosely typed language**", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (e.g. numbers, strings, arrays, etc).

For example, if you declare a variable and give it a value encapsulated in quotes, the browser will treat the variable as a string:

```
var myString = 'Hello';
```

It will still be a string, even if it contains numbers, so be careful:

```
var myNumber = '500'; // oops, this is still a string
typeof(myNumber);
myNumber = 500; // much better — now this is a number
typeof(myNumber);
```

You'll notice that we are using a special function called typeof() — this returns the data type of the variable you pass into it. The first time it is called, it should return string, as at that point the myNumber variable contains a string, '500'.

Numbers versus string

- So what happens when we try to add (or concatenate) a string and a number? Let's try it in our console:

  ```
  'Front ' + 242;
  ```

You might expect this to throw an error, but it works just fine. Trying to represent a string as a number doesn't really make sense, but representing a number as a string does, so the browser rather cleverly converts the number to a string and concatenates the two strings together.

- You can even do this with two numbers — you can force a number to become a string by wrapping it in quote marks.

  ```
  var myDate = '19' + '67';
  typeof myDate;
  ```

If you have a numeric variable that you want to convert to a string but not change otherwise, or a string variable that you want to convert to a number but not change otherwise, you can use the following two constructs:

- The Number object will convert anything passed to it into a number, if it can. Try the following:

```
var myString = '123';
var myNum = Number(myString);
typeof myNum; // number
```

```
var myString = 'hello';
var myNum = Number(myString);
typeof myNum; // NaN
```

- On the other hand, every number has a method called toString() that will convert it to the equivalent string. Try this:

  ```
  var myNum = 123;
  var myString = myNum.toString();
  typeof myString;
  ```

Finding a substring inside a string and extracting it

Sometimes you'll want to find if a smaller string is present inside a larger one (we generally say *if a substring is present inside a string*). This can be done using the indexOf() method, which takes a single parameter — the substring you want to search for. Try this:

- `browserType.indexOf('zilla');` // return 2 (because substring “zilla” starts at position 2)
- browserType.indexOf('vanilla'); //This should give you a result of -1 — this is returned when the substring, in this case 'vanilla', is not found in the main string.

- When you know where a substring starts inside a string, and you know at which character you want it to end, slice() can be used to extract it. Try the following:

  ```
  browserType.slice(0,3); // returns moz
  ```

- Also, if you know that you want to extract all of the remaining characters in a string after a certain character, you don't have to include the second parameter! Instead, you only need to include the character position from where you want to extract the remaining characters in a string. Try the following:

  ```
  browserType.slice(2); // This returns “zilla”
  ```

Changing case

The string methods toLowerCase() and toUpperCase() take a string and convert all the characters to lower- or uppercase, respectively.

```
var radData = 'My NaMe Is MuD';
radData.toLowerCase();
radData.toUpperCase();
```

Updating parts of a string

You can replace one substring inside a string with another substring using the replace() method. This works very simply at a basic level, although there are some advanced things you can do with it that we won't go into yet.

It takes two parameters — the string you want to replace, and the string you want to replace it with.

```
browserType.replace('moz','van');
```

Creating an array

Arrays are constructed of square brackets, which contain a list of items separated by commas.

1. Let's say we wanted to store a shopping list in an array — we'd do something like the following. Enter the following lines into your console:

```
var shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
```

Note : We can store any item in an array — string, number, object, another variable, even another array. You can also mix and match item types — they don't all have to be numbers, strings, etc.

```
 var sequence = [8,4,5,6];
var random = ['tree', 795, [0, 1, 2]]; // accession first item random[0]
```

Finding the length of an array:

```
  var random= [‘tree’,4,5,6];
 random.length;
```

Converting between strings and arrays

Let's play with this, to see how it works. First, create a string in your console:

1. `var myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';`
2. `var myArray = myData.split(',');  // ["Manchester", "London", "Liverpool", "Birmingham", "Leeds", "Carlisle"]`

3. Var myArray = myData.split(); //
   ["Manchester,London,Liverpool,Birmingham,Leeds,Carlisle"]

You can also form the string from array using the join() method. Try the following:

```
var myNewString = myArray.join(',');
myNewString;
```

Another way of converting an array to a string is to use the toString() method. toString() is arguably simpler than join() as it doesn't take a parameter, but more limiting. With join() you can specify different separators (try running Step 4 with a different character than a comma).

```
var dogNames = ["Rocket","Flash","Bella","Slugger"];
dogNames.toString(); //Rocket,Flash,Bella,Slugger
```

Adding and removing array items

We've not yet covered adding and removing array items — let us look at this now. We'll use the myArray array we ended up with in the last section. If you've not already followed that section, create the array first in your console:

```
var myArray = ['Manchester', 'London', 'Liverpool', 'Birmingham', 'Leeds', 'Carlisle'];
```

First of all, to add or remove an item at the end of an array we can use push() and pop() respectively.

- Let's use push() first — note that you need to include one or more items that you want to add to the end of your array. Try this:

```
myArray.push('Cardiff');
myArray;
myArray.push('Bradford', 'Brighton');
myArray;
```

- The new length of the array is returned when the method call completes. If you wanted to store the new array length in a variable, you could do something like this:

```
var newLength = myArray.push('Bristol');
myArray;
newLength;
```

- Removing the last item from the array is as simple as running pop() on it. Try this:

1. myArray.pop();
2. The item that was removed is returned when the method call completes. You could also do this:
3. var removedItem = myArray.pop();
   myArray;
   removedItem;

unshift() and shift() work in exactly the same way, except that they work on the beginning of the array, not the end.

1. First unshift() — try the following commands:

2. myArray.unshift('Edinburgh');
   myArray;
3. Now shift(); try these!
4. var removedItem = myArray.shift();
   myArray;
   removedItem;

Array:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Instanceof:

The instanceof operator returns true if the specified object is of the specified object type. The syntax is:

```
objectName instanceof objectType
```

where objectName is the name of the object to compare to objectType, and objectType is an object type, such as Date or Array.

Use instanceof when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

For example, the following code uses instanceof to determine whether theDay is a Date object. Because theDay is a Date object, the statements in the ifstatement execute.

```
var theDay = new Date(1995, 12, 17);

typeof theDay // object
if (theDay instanceof Date) {
  // statements to execute
}
```

If..Else statement :

```
var shoppingDone = false;

if (shoppingDone === true) {
  var childsAllowance = 10;
} else {
  var childsAllowance = 5;
}
```

If.else if:

```
var select = document.querySelector('select');
var para = document.querySelector('p');

select.addEventListener('change', setWeather);

function setWeather() {
  var choice = select.value;

  if (choice === 'sunny') {
    para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the beach,
or the park, and get an ice cream.';
  } else if (choice === 'rainy') {
    para.textContent = 'Rain is falling outside; take a rain coat and a brolly, and don\'t
stay out for too long.';
```

```
  } else if (choice === 'snowing') {
    para.textContent = 'The snow is coming down — it is freezing! Best to stay in with a
cup of hot chocolate, or go build a snowman.';
  } else if (choice === 'overcast') {
    para.textContent = 'It isn\'t raining, but the sky is grey and gloomy; it could turn any
minute, so take a rain coat just in case.';
  } else {
    para.textContent = '';
  }
}
```

Logical operators: AND, OR and NOT

```
if (choice === 'sunny' && temperature < 86) {
  para.textContent = 'It is ' + temperature + ' degrees outside — nice and sunny. Let\'s
go out to the beach, or the park, and get an ice cream.';
} else if (choice === 'sunny' && temperature >= 86) {
  para.textContent = 'It is ' + temperature + ' degrees outside — REALLY HOT! If you
want to go outside, make sure to put some suncream on.';
}
```

Switch statement:

```
var select = document.querySelector('select');
var para = document.querySelector('p');

select.addEventListener('change', setWeather);

function setWeather() {
  var choice = select.value;
  switch (choice) {
    case 'sunny':
      para.textContent = 'It is nice and sunny outside today. Wear shorts! Go to the
beach, or the park, and get an ice cream.';
      break;
    case 'rainy':
      para.textContent = 'Rain is falling outside; take a rain coat and a brolly, and don\'t
stay out for too long.';
      break;
    case 'snowing':
      para.textContent = 'The snow is coming down — it is freezing! Best to stay in with a
cup of hot chocolate, or go build a snowman.';
      break;
    case 'overcast':
      para.textContent = 'It isn\'t raining, but the sky is grey and gloomy; it could turn any
minute, so take a rain coat just in case.';
      break;
    default:
      para.textContent = '';
  }
}
```

Ternary Operator:

There is one final bit of syntax we want to introduce you to, before we get you to play with some examples. The ternary or conditional operator is a small bit of syntax that tests a condition and returns one value/expression if it is true, and another if it is false — this can be useful in some situations, and can take up a lot less code than an if...elseblock if you simply have two choices that are chosen between via a true/falsecondition.

```
var greeting = ( isBirthday ) ? 'Happy birthday Mrs. Smith — we hope you have a great day!' : 'Good morning Mrs. Smith.';
```

Ex:2

```
<label for="theme">Select theme: </label>
<select id="theme">
  <option value="white">White</option>
  <option value="black">Black</option>
</select>

<h1>This is my website</h1>
```

Java script:

```
var select = document.querySelector('select');
var html = document.querySelector('html');
document.body.style.padding = '10px';

function update(bgColor, textColor) {
  html.style.backgroundColor = bgColor;
  html.style.color = textColor;
}

select.onchange = function() {
  ( select.value === 'black' ) ? update('black','white') : update('white','black');
}
```

For loop:

```
var cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
var info = 'My cats are called ';
var para = document.querySelector('p');

for (var i = 0; i < cats.length; i++) {
  info += cats[i] + ', ';
}

para.textContent = info;
```

Note : We can exit the loop using ‘break’ statement.

While loop:

```
var i = 0;

while (i < cats.length) {
  if (i === cats.length - 1) {
    info += 'and ' + cats[i] + '.';
  } else {
    info += cats[i] + ', ';
  }

  i++;
}
```

labeled Statement

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

break statement

Use the break statement to terminate a loop, switch, or in conjunction with a labeled statement.

- When you use break without a label, it terminates the innermost enclosing while, do-while, for, or switch immediately and transfers control to the following statement.
- When you use break with a label, it terminates the specified labeled statement.

```
var x = 0;
var z = 0;
labelCancelLoops: while (true) {
  console.log('Outer loops: ' + x);
  x += 1;
  z = 1;
  while (true) {
    console.log('Inner loops: ' + z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
}
```

continue statement

The continue statement can be used to restart a while, do-while, for, or label statement.

- When you use continue without a label, it terminates the current iteration of the innermost enclosing while, do-while, or for statement and continues execution of the loop with the next iteration. In contrast to the break statement, continue does not terminate the execution of the loop entirely. In a while loop, it jumps back to the condition. In a for loop, it jumps to the increment-expression.
- When you use continue with a label, it applies to the looping statement identified with that label.

Example 1:

```
var i = 0;
var n = 0;
while (i < 5) {
  i++;
  if (i == 3) {
    continue;
  }
  n += i;
}
```

Example 2:

```
var i = 0;
var j = 10;
checkiandj:
  while (i < 4) {
    console.log(i);
    i += 1;
    checkj:
      while (j > 4) {
        console.log(j);
        j -= 1;
        if ((j % 2) == 0) {
          continue checkj;
        }
        console.log(j + ' is odd.');
      }
      console.log('i = ' + i);
      console.log('j = ' + j);
  }
```

for .. in statement

The for...in statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements. A for...in statement looks as follows:

```
var person = {fname:"John", lname:"Doe", age:25};
var text = "";
var x;
for (x in person) {
	console.log(x)
	console.log(person[x])
}
```

o/p:

```
fname
John
lname
Doe
age
 25
```

for.. of statement

The for...of statement creates a loop Iterating over iterable objects (including Array, Map, Set, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

The following example shows the difference between a for...of loop and a for...in loop. While for...in iterates over property names, for...ofiterates over property values:

```
var arr = [3, 5, 7];
arr.foo = 'hello';

for ( i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for ( i of arr) {
   console.log(i); // logs 3, 5, 7
}
```

Functions vs Methods:

One thing we need to clear up before we move on — technically speaking, **built in browser functions are not functions — they are methods.** This sounds a bit scary and confusing, but don't worry — the words function and method are largely interchangeable, at least for our purposes, at this stage in your learning.

The distinction is that methods are functions defined inside objects. Built-in browser functions (**methods**) and variables (which are called **properties**) are stored inside structured objects, to make the code more efficient and easier to handle.

Invoking functions:

```
function square(number) {
  return number * number;
}
```

```
square(5)
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

var mycar = {make: 'Honda', model: 'Accord', year: 1998};
var x, y;

x = mycar.make; // x gets the value "Honda"
```

```
myFunc(mycar); // calling function
y = mycar.make; // y gets the value "Toyota"
```

Built-in browser functions:

Every,time we manipulated an array,

```
var myArray = ['I', 'love', 'chocolate', 'frogs'];
var madeAString = myArray.join(' ');
console.log(madeAString); // I love chocolate frogs
```

Anonymous function:

We can also create function without a name.

Example 1:

```
function() {
  alert('hello');
}
```

Example 2:

```
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };
console.log(factorial(3));
```

This is called an **anonymous function** — it has no name! It also won't do anything on its own. You generally use an anonymous function along with an event handler, for example the following would run the code inside the function whenever the associated button is clicked:

```
var myButton = document.querySelector('button');


myButton.onclick = function() {
  alert('hello');
}
```

You can also assign an anonymous function to be the value of a variable, for example:

```
var myGreeting = function() {
  alert('hello');
}

myGreeting ;
```

Function scope and conflicts:

- Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- Also, **a function defined in the global scope can access all variables defined in the global scope**. A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

Example 1:

```
// The following variables are defined in the global scope
var num1 = 20,
    num2 = 3,
    name = 'Chamahk';

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore() {
  var num1 = 2, num2 = 3;

  function add() {
    return name + ' scored ' + (num1 + num2);
  }

  return add();
}

getScore(); // Returns "Chamahk scored 5"
```

Example 2:

```
function myBigFunction() {
  var myValue = 1;// to access this variable inside subFunctions, we need to pass this
variable as a parameter

  subFunction(myValue);
}

function subFunction(value) {
  console.log(value);
}
```

Nested functions & closure :

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*. A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

To summarize:

- The inner function can be accessed only from statements in the outer function.

- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // returns 13
b = addSquares(3, 4); // returns 25
c = addSquares(4, 5); // returns 41
```

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3); // Think of it like: give me a function that adds 3 to whatever you
give
                        // it
result = fn_inside(5); // returns 8 becoz, x=3 and y =5
```

```
result1 = outside(3)(5); // returns 8
```

Preservation of variables:

Notice how x is preserved when inside is returned. A closure must preserve the arguments and variables in all scopes it references. Since each call provides potentially different arguments, a new closure is created for each call to outside. The memory can be freed only when the returned inside is no longer accessible.

Logic behind nested function :

```
function A (x) {
   function B(y){
      function C (z) {
         return x + y + z ;
      }
      return C;
   }
   return B;
}
```

Here, In this example, C accesses B's y and A's x. This can be done because:

1. B forms a closure including A, i.e. B can access A's arguments and variables.
2. C forms a closure including B.
3. Because B's closure includes A, C's closure includes A, C can access both B*and* A's arguments and variables. In other words, C *chains* the scopes of Band A in that order.

Easy access to assign values to x, y, z is

```
result_Var = A(1)(2)(3) // returns 6
```

Or else , step by step

```
assign_y_value = A(100)
// returns
ƒ B(y){
      function C (z) {
         return 100 + y + z ;
      }
      return C;
   }
```

Step 2:

```
assign_z_value = assign_y_value(200)
//returns
ƒ C(z) {
         return 100 + 200 + z ;
      }
```

Step 3:

```
result = assign_z_value(300)
// returns 600
```

Name conflicts :

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More inner scopes take precedence, so the inner-most scope takes the highest precedence, while the outer-most scope takes the lowest. This is the scope chain. The first on the chain is the inner-most scope, and the last is the outer-most scope. Consider the following:

Ex: 1

```
function outside() {
  var x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;
}

outside()(10); // returns 20 instead of 10
```

Ex:2

```
function outside() {
  var x = 5;
  function inside(y) {
    return x * 2;
  }
  return inside;
}

outside()(10); // returns 10
```

The name conflict happens at the statement return x and is between inside's parameter x and outside's variable x. The scope chain here is {inside, outside, global object}. Therefore inside's x takes precedences over outside's x, and 20 (inside's x) is returned instead of 10 (outside's x).

Emulating private methods with closures

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code: they also provide a powerful way of managing your global namespace, keeping non-essential methods from cluttering up the public interface to your code.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Using closures in this way is known as the module pattern.

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
```

```
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```

Here, though, we create a single lexical environment that is shared by three functions: counter.increment, counter.decrement, and counter.value.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The lexical environment contains two private items: a variable called `privateCounter` and a function called `changeBy`. Neither of these private items can be accessed directly from outside the anonymous function. Instead, they must be accessed by the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and `changeBy` function.

You'll notice we're defining an anonymous function that creates a counter, and then we call it immediately and assign the result to the `counter` variable. We could store this function in a separate variable `makeCounter` and use it to create several counters.

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();
alert(counter1.value()); /* Alerts 0 */
counter1.increment();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Notice how each of the two counters, counter1 and counter2, maintains its independence from the other. Each closure references a different version of the privateCounter variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable; however changes to the variable value in one closure do not affect the value in the other closure.

Closure Scope Chain

For every closure we have three scopes

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

So, we have access to all three scopes for a closure but often make a common mistake when we have nested inner functions. Consider the following example:

```
// global scope
var e = 10;
function sum(a){
  return function(b){
    return function(c){
      // outer functions scope
      return function(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}

console.log(sum(1)(2)(3)(4)); // log 20

// We can also write without anonymous functions:

// global scope
var e = 10;
function sum(a){
  return function sum2(b){
    return function sum3(c){
      // outer functions scope
      return function sum4(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}

var s = sum(1);
var s1 = s(2);
var s2 = s1(3);
var s3 = s2(4);
console.log(s3) //log 20
```

Creating closures in loops: A common mistake

Prior to the introduction of the let keyword in ECMAScript 2015, a common problem with closures occurred when they were created inside a loop. Consider the following example:

Html

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
```

JS

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to onfocus are closures; they consist of the function definition and the captured environment from the setupHelp function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (item.help). The value of item.help is determined when the onfocus callbacks are executed. Because the loop has already run its course by that time, the item variable object (shared by all three closures) has been left pointing to the last entry in the helpText list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}
```

```
function makeHelpCallback(help) {
  return function() {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the `makeHelpCallback` function creates *a new lexical environment* for each callback, in which `help` refers to the corresponding string from the `helpText` array.

If you don't want to use more closures, you can use the let keyword introduced in ES2015 :

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    let item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

This example uses `let` instead of `var`, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Using the argument's object:

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

`arguments[i]`

where i is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by arguments.length.

Using the arguments object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use arguments.length to determine the number of arguments actually passed to the function, and then access each argument using the arguments object.

```
function myConcat(separator) {
   var result = ' ’ ;   // initialize list
   var i;
   // iterate through arguments
   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}
```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');
 // returns "elephant; giraffe; lion; cheetah; "
```

Function parameter:

Starting with ECMAScript 2015, there are two new kinds of parameters: default parameters and rest parameters.

Default parameter:

In JavaScript, parameters of functions default to undefined. However, in some situations it might be useful to set a different default value. This is where default parameters can help.

In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are undefined. If in the following example, no value is provided for b in the call, its value would be undefined when evaluating a*b and the call to multiply would have returned NaN. However, this is caught with the second line in this example:

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;
  return a * b;
}

multiply(5); // 5
```

With default parameters, the check in the function body is no longer necessary. Now, you can simply put 1 as the default value for b in the function head:

```
function multiply(a, b = 1) {
  return a * b;
}
multiply(5); // 5

multiply(5,6); // 30
```

Rest parameter:

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. In the example, we use the rest parameters to collect arguments from the second one to the end. We then multiply them by the first one. This example is using an arrow function, which is introduced in the next section.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Arrow functions:

An arrow function expression (previously, and now incorrectly known as fat arrow function) has a shorter syntax compared to function expressions and does not have its own this, arguments, super, or new.target. **Arrow functions are always anonymous**.

Two factors influenced the introduction of arrow functions: shorter functions and non-binding of `this`.

- **Shorter function**

In some functional patterns, shorter functions are welcome. Compare:

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

var a2 = a.map(function(s) { return s.length; });
```

```
console.log(a2); // logs [8, 6, 7, 9]

var a3 = a.map(s => s.length);
console.log(a3); // logs [8, 6, 7, 9]
```

No separate this

```
function Person() {
  // The Person() constructor defines `this` as itself.
  this.age = 0;

  setInterval(function growUp() {
    // In nonstrict mode, the growUp() function defines `this`
    // as the global object, which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}

var p = new Person();
```

In ECMAScript 3/5, this issue was fixed by assigning the value in this to a variable that could be closed over.

```
function Person() {
  var self = this; // Some choose `that` instead of `self`.
                   // Choose one and be consistent.
  self.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `self` variable of which
    // the value is the expected object.
    self.age++;
  }, 1000);
}
```

An arrow function does not have its own this; the **this value of the enclosing execution context is used.** Thus, in the following code, the this within the function that is passed to setInterval has the same value as this in the enclosing function:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

CallBack Vs Promise

CallBack:

```
function first ( value, callback) {
    value = value +2 ;
    callback(value, false) ;
}
```

```
function second ( value, callback) {
   value = value +2 ;
   callback(value, false) ;
}

function third ( value, callback) {
   value = value +2 ;
   callback(value, false) ;
}

function fourth ( value) {
   console.log('..... final result .....');
   console.log(value);
}

first( 2, function (firstResult, err) {
        if ( ! err) {
                second ( firstResult, function (secondResult, err) {
                        if ( ! err) {
                                third( secondResult, function(thirdResult, err) {
                                        if (!err) {
                                                fourth(thirdResult);
                                        }
                                });
                        }
                });
        }
});
```

Promise :

```
function first ( value) {
    return value +2 ;
}

 function second ( value) {
    return value +2 ;
}

 function third ( value) {
    return value +2 ;
}

var promise = new Promise ( function(resolve, reject) {
          resolve (2)
})

promise.then(first).then(second).then(third).then( function (response) {
console.log(response)
})
```

Promise:

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Promise states:

A promise can be in one of three states.

- **Pending** - the promise’s outcome hasn’t yet been determined, because the asynchronous operation that will produce its result hasn’t completed yet.
- **Fulfilled** - the asynchronous operation has completed, and the promise has a value.
- **Rejected** - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.

When a promise is pending, it can transition to the fulfilled or rejected state.

Using promises:

The primary API for a promise is its `then` method, which registers callbacks to receive either the eventual value or the reason why the promise cannot be fulfilled. Here is a simple “hello wolrd” program that synchronously obtains and logs a greeting.

```
var greeting = sayHello();
console.log(greeting);    // 'hello world’
```

However, if `sayHello` is asynchronous and needs to look up the current greeting from a web service, it may return a promise:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'hello world’
});
```

The same message is printed to the console, but now other code can continue while the greeting is being fetched.

As mentioned above, a promise can also represent a failure. If the network goes down and the greeting can’t be fetched from the web service, you can register to handle the failure using the second argument to the promise’s then method:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
   console.log(greeting);    // 'hello world’
}, function (error) {
   console.error('uh oh: ', error);   // 'uh oh: something bad happened’
});
```

If sayHello succeeds, the greeting will be logged, but if it fails, then the reason, i.e. error, will be logged using console.error.

Transforming Future values:

One powerful aspect of promises is allowing you to transform future values by returning a new value from callback function passed to then. For example:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
   return greeting + '!!!!';
}).then(function (greeting) {
   console.log(greeting);    // 'hello world!!!!’
});
```

Sequencing asynchronous operations:

A function passed to then can also return another promise. This allows asynchronous operations to be chained together, so that they are guaranteed to happen in the correct order. For example, if addExclamation is asynchronous (possibly needing to access another web service and returns a promise for the new greeting:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    return addExclamation(greeting); // addExclamation returns a promise
}).then(function (greeting) {
    console.log(greeting);    // 'hello world!!!!’
});
```

This can be written more simply as:

```
var greetingPromise = sayHello();
greetingPromise
    .then(addExclamation)
    .then(function (greeting) {
        console.log(greeting);    // 'hello world!!!!’
    });
```

Handling Errors:

What if an error occurs while performing an asynchronous operation? For example, what if sayHello, or addExclamation fails? In synchronous code, you can use try/catch, and rely on exception propagation to handle errors in one spot. Here is a synchronous version of the previous example that include try/catch error handling. If an error occurs in either sayHello or addExclamation, the catch block will be executed.

```
var greeting;
try {
    greeting = sayHello();
    greeting = addExclamation(greeting);
    console.log(greeting);    // 'hello world!!!!’
} catch(error) {
    console.error('uh oh: ', error);   // 'uh oh: something bad happened’
}
```

When dealing with asynchronous operations, synchronous try/catch can no longer be used. However, promises allow handling asynchronous errors in a very similar way. This allows you not only to write asynchronous code that is similar in style to synchronous code, but also to reason about asynchronous control flow and error handling similarly to synchronous code.

Here is an asynchronous version that handles errors in the same way:

```
var greetingPromise = sayHello();
greetingPromise
    .then(addExclamation)
    .then(function (greeting) {
        console.log(greeting);    // 'hello world!!!!’
    }, function(error) {
        console.error('uh oh: ', error);   // 'uh oh: something bad happened’
    });
```

Advantage of Promise over traditional callbacks:

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with then() even *after* the success or failure of the asynchronous operation, will be called, as above.
- **Multiple callbacks** may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted.

One of the great things about using promises is **chaining**.

Chaining:

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**.

Here's the magic: the then() function returns a new promise, different from the original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

or

```
const promise2 = doSomething().then(successCallback, failureCallback);
```

This second promise represents the completion not just of doSomething(), but also of the successCallback or failureCallback you passed in, which can be other asynchronous functions returning a promise. When that's the case, any callbacks added to promise2 get queued behind the promise returned by either successCallback or failureCallback.

Basically, each promise represents the completion of another asynchronous step in the chain.

In the old days, doing several asynchronous operations in a row would lead to the classic callback pyramid of doom:

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

With modern functions, we attach our callbacks to the returned promises instead, forming a promise chain:

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(failureCallback);
```

The arguments to then are optional, and catch(failureCallback) is short for then(null, failureCallback). You might see this expressed with arrow functions instead:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

Important: Always return results, otherwise callbacks won't catch the result of a previous promise.

Chaining after a catch:

It's possible to chain *after* a failure, i.e. a catch, which is useful to accomplish new actions even after an action failed in the chain. Read the following example:

```
new Promise((resolve, reject) => {
    console.log('Initial');

    resolve();
})
.then(() => {
    throw new Error('Something failed');

    console.log('Do this');
})
.catch(() => {
    console.log('Do that');
})
.then(() => {
    console.log('Do this, no matter what happened before');
});
```

This will output the following text:

```
Initial
Do that
Do this, no matter what happened before
```

Note that the text `Do this` is not output because the `Something failed` error caused a rejection.

Error propagation:

You might recall seeing failureCallback three times in the pyramid of doom earlier, compared to only once at the end of the promise chain:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => console.log(`Got the final result: ${finalResult}`))
.catch(failureCallback);
```

Basically, a promise chain stops if there's an exception, looking down the chain for catch handlers instead. This is very much modeled after how synchronous code works:

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

This symmetry with synchronous code culminates in the async/await syntactic sugar in ECMAScript 2017:

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch(error) {
    failureCallback(error);
  }
}
```

Promise example Code:

To get started, let’s examine the following code, which creates a new Promiseobject:

```
const promise = new Promise((resolve, reject) => {
  //asynchronous code goes here
});
```

We start by instantiating a new Promise object and passing it a callback function. The callback takes two arguments, resolve and reject, which are both functions. All your asynchronous code goes inside that callback. If everything is successful, the promise is fulfilled by calling resolve(). In case of an error, reject() is called with an Error object. This indicates that the promise is rejected.

Now let’s build something simple which shows how promises are used. The following code makes an asynchronous request to a web service that returns a random joke in JSON format. Let’s examine how promises are used here:

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.icndb.com/jokes/random');
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response); // we got data here, so resolve the Promise
    } else {
      reject(Error(request.statusText)); // status is not 200 OK, so reject
    }
  };

  request.onerror = () => {
    reject(Error('Error fetching data.')); // error occurred, reject the  Promise
  };

  request.send(); // send the request
});

console.log('Asynchronous request made.');

promise.then((data) => {
  console.log('Got data! Promise fulfilled.');
  document.body.textContent = JSON.parse(data).value.joke;
}, (error) => {
  console.log('Promise rejected.');
  console.log(error.message);
});
```

In the previous code, the Promise constructor callback contains the asynchronous code used to get data the from remote service. Here, we just create an Ajax request to https://api.icndb.com/jokes/random, which returns a random joke. When a JSON response is received from the remote server, it's passed to resolve(). In case of any error, reject() is called with an Error object.

When we instantiate a Promise object, we get a proxy to the data that will be available in future. In our case, we're expecting some data to be returned from the remote service at some point in future. So, how do we know when the data becomes available? This is where the Promise.then() function is used. This function takes two arguments: a success callback and a failure callback. These callbacks are called when the Promise is settled (i.e. either fulfilled or rejected). If the promise was fulfilled, the success callback will be fired with the actual data you passed to `resolve()`. If the promise was rejected, the failure callback will be called. Whatever you passed to `reject()` will be passed as an argument to this callback.

Async

Let's start with the `async` keyword. It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

The word "async" before a function means one simple thing: a function always returns a promise. Even If a function actually returns a non-promise value, prepending the function definition with the "async" keyword directs JavaScript to automatically wrap that value in a resolved promise.

For instance, the code above returns a resolved promise with the result of `1`, let's test it:

```
async function f() {
  return 1;
}
f().then(alert); // alert 1
```

We could explicitly return a promise, that would be the same as:

```
async function f() {
  return Promise.resolve(1);
}
f().then(alert); // 1
```

Await

There’s another keyword, await, that works only inside async functions, and it’s pretty cool.

```
let value = await promise;
```

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here’s an example with a promise that resolves in 5 second

```
async function f() {
  let result = await new Promise((resolve, reject) => { // wait till promise resolves *
    setTimeout(() => resolve("done!"), 5000)
  });
  alert(result); // "done!"
}
f();
```

The function execution “pauses” at the line (*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in five second.

It’s just a more elegant syntax of getting the promise result than promise.then, easier to read and write.

Note: If we try to use await in non-async function, there would be a syntax error:

await won’t work in the top-level code

People who are just starting to use await tend to forget the fact that we can’t use await in top-level code. For example, this will not work:

```
// syntax error in top-level code
```

```
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

We can wrap it into an anonymous async function, like this:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Error handling

If a promise resolves normally, then `await promise` returns the result. But in case of a rejection, it throws the error, just as if there were a `throw` statement at that line.

This code:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

is the same as this:

```
async function f() {
  throw new Error("Whoops!");
}
```

In real situations, the promise may take some time before it rejects. So `await` will wait, and then throw an error.

We can catch that error using `try..catch`, the same way as a regular `throw`:

```
async function f() {
  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

In case of an error, the control jumps to the `catch` block. We can also wrap multiple lines:

```
async function f() {
  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

f();
```

If we don’t have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
async function f() {
  let response = await fetch('http://no-such-url');

}
```

```
// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add .catch there, then we get an unhandled promise error (viewable in the console).

async/await and promise.then/catch

When we use async/await, we rarely need .then, because await handles the waiting for us. And we can use a regular try..catch instead of .catch. That's usually (not always) more convenient.

But at the top level of the code, when we're outside of any async function, we're syntactically unable to use await, so it's a normal practice to add .then/catch to handle the final result or falling-through errors.

Like in the line (*) of the example above.

async/await works well with Promise.all

When we need to wait for multiple promises, we can wrap them in Promise.all and then await:

```
// wait for the array of results
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
  ]);
```

Iterators:

A object is considered *iterable* if it has a method whose key is the symbol Symbol.iterator that returns a so-called *iterator*. The iterator is an object that returns values via its method next(). We say: it *iterates over* the *items* (the content) of the iterable, one per method call.

An iterator is any object which implements the Iterator protocol by having a next() method which returns an object with two properties: value, the next value in the sequence; and done, which is true if the last value in the sequence has already been consumed.

The most common iterator in Javascript is the Array iterator, which simply returns each value in the associated array in sequence.

Purpose of user defined Iterators:

Imagine that you have this array —

```
const myFavouriteAuthors = [
 'Neal Stephenson',
 'Arthur Clarke',
 'Isaac Asimov',
 'Robert Heinlein'
];
```

At some point, you will want to get back all the individual values in the array for printing them on the screen, manipulating them, or for performing some action on them. If I ask you how would you do that? You'll say — *it's easy. I'll just loop over them using for, while, for-of or one of these looping methods.*

Now, imagine that instead of the previous array, you had a custom data structure to hold all your authors. Like this —

```
const myFavouriteAuthors = {
        allAuthors: {
        fiction: [‘J.k.Rowling’, ‘Sujatha’, ‘Krishnan’  ],
        scienceFiction: [‘Isaac’,’Author’,’Neal’],
        fantasy: [‘J.k.Rowling’,’Terry Pretchatt’, ‘J.R.R.Towrien’]
        }
}
```

Now, myFavouriteAuthors is an object which contains another object allAuthors. allAuthors contains three arrays with keys fiction, scienceFiction, and fantasy. Now, if I ask you to loop over myFavouriteAuthors to get all the authors, what would your approach be? You can go ahead and try some combination of loops to get all the data. However if you do this,

```
for (let author of myFavouriteAuthors) {
 console.log(author)
}
// TypeError: {} is not iterable
```

You would get a TypeError saying that the object is not *iterable*. Let’s see what iterables are and how we can make an object iterable. At the end of this article, you’ll know how to use for-of loop on custom objects, and in this case, on myFavouriteAuthors.

Making objects iterable

So as we learnt in the previous section, we need to implement a method called **Symbol.iterator.**

```
const myFavouriteAuthors = {
  allAuthors: {
    fiction: [
      'Agatha Christie',
      'J. K. Rowling',
      'Dr. Seuss'
    ],
    scienceFiction: [
      'Neal Stephenson',
      'Arthur Clarke',
      'Isaac Asimov',
      'Robert Heinlein'
    ],
    fantasy: [
      'J. R. R. Tolkien',
      'J. K. Rowling',
      'Terry Pratchett'
    ],
  },
  [Symbol.iterator]() {
    // Get all the authors in an array
    const genres = Object.values(this.allAuthors);
    // Store the current genre and author index
    let currentGenreIndex = 0;
    let currentAuthorIndex = 0;

    return {
```

```
  // Implementation of next()
  next() {
    // authors according to current genre index
    const authors = genres[currentGenreIndex];

    // doNotHaveMoreAuthors is true when the authors array is exhausted.
    // That is, all items are consumed.
    const doNothaveMoreAuthors = !(currentAuthorIndex < authors.length);

    if (doNothaveMoreAuthors) {
     // When that happens, we move the genre index to the next genre
      currentGenreIndex++;
      // and reset the author index to 0 again to get new set of authors
      currentAuthorIndex = 0;
    }

    // if all genres are over, then we need tell the iterator that we
    // can not give more values.
    const doNotHaveMoreGenres = !(currentGenreIndex < genres.length);
    if (doNotHaveMoreGenres) {
      // Hence, we return done as true.
      return {
        value: undefined,
        done: true
      };
    }

    // if everything is correct, return the author from the
    // current genre and increment the currentAuthorindex
```

```
        // so next time, the next author can be returned.
        return {
          value: genres[currentGenreIndex][currentAuthorIndex++],
          done: false
        }
      }
    };
  }
};

for (const author of myFavouriteAuthors) {
  console.log(author);
}

console.log(...myFavouriteAuthors)
```

Iterables:

An object is **iterable** if it defines its iteration behavior, such as what values are looped over in a for...of construct. Some built-in types, such as Array or Map, have a default iteration behavior, while other types (such as Object) do not.

In order to be **iterable**, an object must implement the **@@iterator** method, meaning that the object (or one of the objects up its prototype chain) must have a property with a Symbol.iterator key.

It may be possible to iterate over an iterable more than once, or only once. It is up to the programmer to know which is the case. Iterables which can iterate only once (e.g. Generators) customarily return **this** from their **@@iterator** method, where those which can be iterated many times must return a new iterator on each invocation of **@@iterator**.

Arrays, String, etc (and Typed Arrays) are iterables over their elements:

```
for (const x of ['a', 'b']) {
   console.log(x);
}
// Output:
// 'a'
// 'b'
```

Getting one by one value from a built-in array

```
var arr= [‘one’,’two’,’three’]
var itr = arr[Symbol.iterator]()
console.log( itr.next() ) // { value: ‘one’, done: false }
```

Introduction to Events

There are a lot of different types of event that can occur, for example:

- The user clicking the mouse over a certain element, or hovering the cursor over a certain element.
- The user pressing a key on the keyboard.
- The user resizing or closing the browser window.
- A web page finishing loading.
- A form being submitted.
- A video being played, or paused, or finishing play.
- An error occuring.

You will gather from this (and from glancing at the MDN Event reference) that there are **a lot** of events that can be responded to.

Each available event has an **event handler**, which is block of code (usually a user-defined JavaScript function) that will be run when the event fires. When such a block of code is defined to be run in response to an event firing, we say we are **registering an event handler**. Note that event handlers are sometimes called **event listeners** — they are pretty much interchangeable for our purposes, although strictly speaking they work together. The listener listens out for the event happening, and the handler is the code that is run in response to it happening.

Event handler properties:

```
var btn = document.querySelector('button');
var btn = document.querySelector('#button'); // if button is an id
btn.onclick = function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

The onclick property is the event handler property being used in this situation. It is essentially a property like any other available on the button (e.g. btn.textContent, or btn.style), but it is a special type — when you set it to be equal to some code, that code will be run when the event fires on the button.

You could also set the handler property to be equal to a named function name (like we saw in Build your own function). The following would work just the same:

```
var btn = document.querySelector('button');

function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;
```

Some events are very general and available nearly anywhere (for example an onclickhandler can be registered on nearly any element), whereas some are more specific and only useful in certain situations (for example it makes sense to use onplay only on specific elements, such as <video>).

Inline event handlers - (don't use these)

```
<button onclick="alert('Hello, this is my old-fashioned event handler!');">Press
me</button>

<button onclick="bgChange()">Press me</button>

function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

Disadvantage of using inline event handler:

Even in a single file, inline event handlers are not a good idea. One button is OK, but what if you had 100 buttons? You'd have to add 100 attributes to the file; it would very quickly turn into a maintenance nightmare. With JavaScript, you could easily add an event handler function to all the buttons on the page no matter how many there were, using something like this:

```
var buttons = document.querySelectorAll('button');

for (var i = 0; i < buttons.length; i++) {
  buttons[i].onclick = bgChange;
}
```

addEventListener() and removeEventListener()

The newest type of event mechanism is defined in the Document Object Model (DOM) Level 2 Events Specification, which provides browsers with a new function — addEventListener(). This functions in a similar way to the event handler properties, but the syntax is obviously different. We could rewrite our random color example to look like this:

```
var btn = document.querySelector('button');
function bgChange() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
btn.addEventListener('click', bgChange);
```

So inside the `addEventListener()` function, we specify two parameters — the name of the event we want to register this handler for, and the code that comprises the handler function we want to run in response to it. Note that it is perfectly appropriate to put all the code inside the `addEventListener()` function, in an anonymous function, like this:

```
btn.addEventListener('click', function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
});
```

```
btn.removeEventListener('click', bgChange);
```

This isn't significant for simple, small programs, but for larger, more complex programs it can improve efficiency to clean up old unused event handlers, plus this allows you to for example have the same button performing different actions in different circumstances — all you've got to do is add/remove event handlers as appropriate.

Second, you can also register multiple handlers for the same listener. The following two handlers would not be applied:

```
myElement.onclick = functionA;
myElement.onclick = functionB;
```

As the second line would overwrite the first value of `onclick` set. This would work, however:

```
myElement.addEventListener('click', functionA);
myElement.addEventListener('click', functionB);
```

Both functions would now run when the element is clicked.

For ref: `addEventListener()` and `removeEventListener()`reference pages.

What mechanism should I use:

Of the three mechanisms, you definitely shouldn't use the HTML event handler attributes — these are outdated, and bad practice, as mentioned above.

The other two are relatively interchangeable, at least for simple uses:

- Event handler properties have less power and options, but better cross browser compatibility (being supported as far back as Internet Explorer 8). You should probably start with these as you are learning.
- DOM Level 2 Events (`addEventListener()`, etc.) are more powerful, but can also become more complex and are less well supported (supported as far back as Internet Explorer 9). You should also experiment with these, and aim to use them where possible.

The main advantages of the third mechanism are that you can remove event handler code if needed, using `removeEventListener()`, and you can add multiple listeners of the same type to elements if required. For example, you can call `addEventListener('click', function() { ... })` on an element multiple times, with different functions specified in the second argument. This is impossible with event handler properties, because any subsequent attempts to set a property will overwrite earlier ones, e.g.:

```
element.onclick = function1;
element.onclick = function2;
etc.
```

Event Objects:

Sometimes inside an event handler function you might see a parameter specified with a name such as `event`, `evt`, or simply `e`. This is called the event object, and it is automatically passed to event handlers to provide extra features and information. For example, let's rewrite our random color example again slightly:

```
function bgChange(e) {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  e.target.style.backgroundColor = rndCol;
  console.log(e);
}

btn.addEventListener('click', bgChange);
```

Here you can see that we are including an event object, **e**, in the function, and in the function setting a background color style on `e.target` — which is the button itself. The `target` property of the event object is always a reference to the element that the event has just occurred upon. So in this example we are setting a random background color on the button, not the page.

Note: You can use any name you like for the event object — you just need to choose a name that you can then use to reference it inside the event handler function. e/evt/event are most commonly used by developers because they are short and easy to remember. It's always good to stick to a standard.

`e.target` is incredibly useful when you want to set the same event handler on multiple elements, and do something to all of them when an event occurs on them. You might for example have a set of 16 tiles that disappear when they are clicked on.

```
var divs = document.querySelectorAll('div');

for (var i = 0; i < divs.length; i++) {
  divs[i].onclick = function(e) {
    e.target.style.backgroundColor = bgChange();
  }
}
```

Preventing default behaviour:

Sometimes, you'll come across a situation where you want to stop an event doing what it does by default. The most common example is that of a web form, for example a custom registration form. When you fill in the details and press the submit button, the natural behaviour is for the data to be submitted to a specified page on the server for processing, and the browser to be redirected to a "success message" page of some kind (or the same page, if another is not specified.)

The trouble comes when the user has not submitted the data correctly — as a developer, you'll want to stop the submission to the server and give them an error message telling them what's wrong and what needs to be done to put things right.

First, a simple HTML form that requires you to enter your first and last name:

```
<form>
 <div>
   <label for="fname">First name: </label>
   <input id="fname" type="text">
 </div>
 <div>
   <label for="lname">Last name: </label>
   <input id="lname" type="text">
 </div>
 <div>
    <input id="submit" type="submit">
 </div>
</form>
<p></p>
```

we call the preventDefault() function on the event object — which stops the form submission — and then display an error message in the paragraph below our form to tell the user what's wrong:

```
var form = document.querySelector('form');
var fname = document.getElementById('fname');
var lname = document.getElementById('lname');
var submit = document.getElementById('submit');
var para = document.querySelector('p');

form.onsubmit = function(e) {
  if (fname.value === '' || lname.value === '') {
    e.preventDefault();
    para.textContent = 'You need to fill in both names!';
  }
}
```

Event bubbling and capture:

Event bubbling and capture are two mechanisms that describe what happens when two handlers of the same event type are activated on one element.

This is a pretty simple example that shows and hides a <div> with a <video> element inside it:

```
<button>Display video</button>

<div class="hidden">
  <video>
    <source src="rabbit320.mp4" type="video/mp4">
    <source src="rabbit320.webm" type="video/webm">
    <p>Your browser doesn't support HTML5 video. Here is a <a
href="rabbit320.mp4">link to the video</a> instead.</p>
```

```
  </video>
</div>
```

When the <button> is clicked, the video is displayed, by changing the class attribute on the <div> from hidden to showing

```
btn.onclick = function() {
  videoBox.setAttribute('class', 'showing');
}
```

We then add a couple more onclick event handlers — the first one to the <div> and the second one to the <video>. The idea is that when the area of the <div> outside the video is clicked, the box should be hidden again; when the video itself is clicked, the video should start to play.

```
videoBox.onclick = function() {
  videoBox.setAttribute('class', 'hidden');
};

video.onclick = function() {
  video.play();
};
```

But there's a problem — currently when you click the video it starts to play, but it causes the <div> to also be hidden at the same time. This is because the video is inside the <div> — it is part of it — so clicking on the video actually runs *both* the above event handlers.

Bubbling and capturing explained:

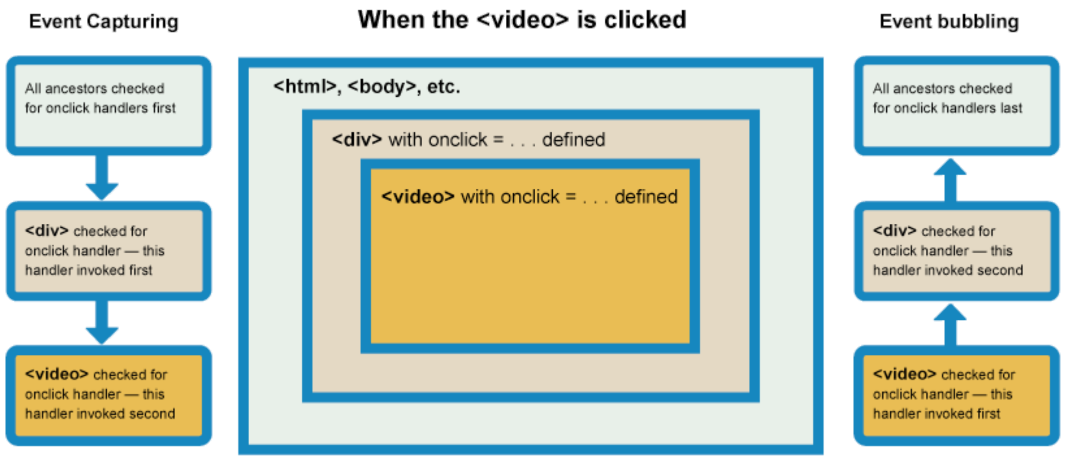
When an event is fired on an element that has parent elements (e.g. the <video> in our case), modern browsers run two different phases — the capturing phase and the bubbling phase.

In the capturing phase:

- The browser checks to see if the element's outer-most ancestor (<html>) has an onclick event handler registered on it in the capturing phase, and runs it if so.
- Then it moves on to the next element inside <html> and does the same thing, then the next one, and so on until it reaches the element that was actually clicked on.

In the bubbling phase, the exact opposite occurs:

- The browser checks to see if the element that was actually clicked on has an onclick event handler registered on it in the bubbling phase, and runs it if so.
- Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the <html> element.


Event Capturing
When the <video> is clicked
Event bubbling
All ancestors checked for onclick handlers first
<div> checked for onclick handler — this handler invoked first
<video> checked for onclick handler — this handler invoked second
<html>, <body>, etc.
<div> with onclick = . . . defined
<video> with onclick = . . . defined
All ancestors checked for onclick handlers last
<div> checked for onclick handler — this handler invoked second
<video> checked for onclick handler — this handler invoked first

In modern browsers, by default, all event handlers are registered in the **bubbling phase**. So in our current example, when you click the video, the click event bubbles from the <video> element outwards to the <html> element. Along the way:

- It finds the video.onclick... handler and runs it, so the video first starts playing.
- It then finds the videoBox.onclick... handler and runs it, so the video is hidden as well.

Fixing the problem with stopPropagation()

This is annoying behaviour, but there is a way to fix it! The standard event object has a function available on it called stopPropagation(), which when invoked on a handler's event object makes it so that handler is run, but the event doesn't bubble any further up the chain, so no more handlers will be run.

We can therefore fix our current problem by changing the second handler function in the previous code block to this:

```
video.onclick = function(e) {
  e.stopPropagation();
  video.play();
};
```

Event delegation:

Bubbling also allows us to take advantage of **event delegation** — this concept relies on the fact that if you want some code to run when you click on any one of a large number of child elements, you can set the event listener on their parent and have the effect of the event listener bubble to each child, rather than having to set the event listener on every child individually.

A good example is a series of list items — if you want each one of them to pop up a message when clicked, you can can set the `click` event listener on the parent `<ul>`, and it will bubble to the list item

This concept is explained further on David Walsh's blog, with multiple examples — see How JavaScript Event Delegation Works.

Objects:

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called **properties** and **methods** when they are inside objects.)

Example :

Creating an object often begins with defining and initializing a variable.

```
var person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' +
this.interests[0] + ' and ' + this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
```

In our person object we've got a string, a number, two arrays, and two functions. The first four items are data items, and are referred to as the object's **properties**. The last two items are functions that allow the object to do something with that data, and are referred to as the object's **methods**.

Dot notation:

We can access the object's properties and methods using **dot notation**. The object name (person) acts as the **namespace** — it must be entered first to access anything **encapsulated** inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
person.age
person.interests[1]
person.bio()
```

```
var foo = {a: 'alpha', 2: 'two'};
console.log(foo.a);    // alpha
console.log(foo[2]);   // two
//console.log(foo.2);  // Error: Unexpected number
//console.log(foo[a]); // Error: a is not defined
console.log(foo['a']); // alpha
console.log(foo['2']); // two
```

Sub-namespaces

It is even possible to make the value of an object member another object. For example, try changing the name member from

```
name: ['Bob', 'Smith'],
```

to

```
name : {
  first: 'Bob',
  last: 'Smith'
},
```

Here we are effectively creating a **sub-namespace**. This sounds complex, but really it's not — to access these items you just need to chain the extra step onto the end with another dot. Try these:

```
person.name.first
person.name.last
```

Important: At this point you'll also need to go through your method code and change any instances of

```
name[0]
name[1]
```

to

```
name.first
name.last
```

Otherwise your methods will no longer work.

Bracket Notation:

There is another way to access object properties — using bracket notation. Instead of using these:

```
person.age
person.name.first
```

You can use

```
person['age']
person['name']['first']
```

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.

Setting object member:

So far we've only looked at retrieving (or **getting**) object members — you can also **set** (update) the value of object members by simply declaring the member you want to set (using dot or bracket notation), like this:

```
person.age = 45;
person['name']['last'] = 'Cratchit';
```

Try entering these lines, and then getting the members again to see how they've changed:

```
person.age
person['name']['last']
```

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these:

```
person['eyes'] = 'hazel';
person.farewell = function() { alert("Bye everybody!"); }
```

Working with objects all along :

When we use string methods like ,

```
myString.split(',');
```

You were using a method available on an instance of the String class. Every time you create a string in your code, that string is automatically created as an instance of String, and therefore has several common methods/properties available on it.

When you accessed the document object model using lines like this:

```
var myDiv = document.createElement('div');
var myVideo = document.querySelector('video');
```

You were using methods available on an instance of the Document class. For each webpage loaded, an instance of Document is created, called document, which represents the entire page's structure, content, and other features such as its URL. Again, this means that it has several common methods/properties available on it.

this keyword

```
const restaurant = {
  name: 'Italian Bistro',
  seatingCapacity: 120,
  hasDineInSpecial: true,
  entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine pesto'],
  openRestaurant() {
    if (hasDineInSpecial) {
      return 'Unlock the door, post the special on the board, then flip the open sign.';
    } else {
      return 'Unlock the door, then flip the open sign.';
    }
  }
};

console.log(restaurant.openRestaurant());
```

The output would be

```
ReferenceError: hasDineInSpecial is not defined
```

The error above doesn't work because hasDineInSpecial is out of the .openRestaurant() method's scope.

To address this scope issue, we can use the `this` keyword to access properties inside of the same object.

```
openRestaurant() {
    if (hasDineInSpecial) {
      return 'Unlock the door, post the special on the board, then flip the open sign.';
    } else {
      return 'Unlock the door, then flip the open sign.';
    }
  }
```

In java script, `this` refers to the object we call it inside.

```
let person = {
  name: 'Gautham',
  sayHello: function() {
    return `Hello, my name is ${this.name}`;
  }
};

let friend = {
  name: "Karthi"
}
friend.sayHello= person.sayHello
console.log(friend.sayHello())
```

Output:

```
Hello, my name is Karthi.
```

It logged your friend's name instead of yours because the meaning of this changed to the friend object, for which the name key is different.

Getters and Setters:

Getter and setter methods get and set the properties inside of an object. There are a couple of advantages to using these methods for getting and setting properties directly:

- You can check if new data is valid before setting a property.
- You can perform an action on the data while you are getting or setting a property.
- You can control which properties can be set and retrieved.

Example:

```
let person = {
  _name: 'Lu Xun',
  _age: 137,

  set age(ageIn) {
    if (typeof ageIn === 'number') {
      this._age = ageIn;
    }
    else {
      console.log('Invalid input');
      return 'Invalid input';
    }
  }
};
```

- We prepended the property names with underscores (_). Developers use an underscore before a property name to indicate a property or value should not be modified directly by other code. We recommend prepending all properties with an underscore, and creating setters for all attributes you want to access later in your code.

- The set age() setter method accepts ageIn as a variable. The ageIn variable holds the new value that we will store in _age.
- Inside of the .age() setter we use a conditional statement to check if the ageIn variable (our new value) is a number.
- If the input value is a number (valid input), then we use this._age to change the value assigned to _age. If it is not valid, then we output a message to the user.

How to call a setter method?

Now that you know how to create a setter method, you may be wondering how we use it. We call setter methods the same way we edited properties.

```
person.age='Thirty-nine'
```

Getter:

Getters are used to get the property values inside of an object.

Getter & Setter code:

```
let person = {
  _name: 'Lu Xun',
  _age: 137,

  set age(ageIn) {
    if (typeof ageIn === 'number') {
      this._age = ageIn;
    }
    else {
      console.log('Invalid input');
      return 'Invalid input';
    }
  },
  get age() {
    console.log(`${this._name} is ${this._age} years old.`);
    return this._age;
  }

};

person.age = 'Thirty-nine';
person.age = 39;

console.log(person.age);
```

Object creation using Constructor:

```
function Person(name) {
  this.name = name;
  this.greeting = function() {
    alert('Hi! I\'m ' + this.name + '.');
  };
}
```

The constructor function is JavaScript's version of a class. You'll notice that it has all the features you'd expect in a function, although it doesn't return anything or explicitly create an object — it basically just defines properties and methods.

Note: A constructor function name usually starts with a capital letter — this convention is used to make constructor functions easier to recognize in code.

So how do we call a constructor to create some objects?

```
var person1 = new Person('Bob');
```

You'll now see that we have two new objects on the page, each of which is stored under a different namespace — when you access their properties and methods, you have to start calls with person1 or person2; they are neatly packaged away so they won't clash with other functionality.

In each case, the new keyword is used to tell the browser we want to create a new object instance, followed by the function name with its required parameters contained in parentheses, and the result is stored in a variable

Note: All Objects are created as if a call to new Object() were made; that is, objects made from object literal expressions are instances of Object.

Deleting properties:

You can remove a non-inherited property by using the delete operator. The following code shows how to remove a property.

```
// Creates a new object, myobj, with two properties, a and b.
var myobj = new Object();
myobj.a = 5;
myobj.b = 12;

// Removes the a property, leaving myobj with only the b property.
delete myobj.a;
console.log ('a' in myobj); // yields "false"
```

You can also use delete to delete a global variable if the var keyword was not used to declare the variable:

```
g = 17;
delete g;
```

Object creation using Object() constructor:

First of all, you can use the Object() constructor to create a new object. Yes, even generic objects have a constructor, which generates an empty object.

```
var person1 = new Object();
```

This stores an empty object in the person1 variable. You can then add properties and methods to this object using dot or bracket notation as desired; try these examples:

```
person1.name = 'Chris';
person1['age'] = 38;
person1.greeting = function() {
  alert('Hi! I\'m ' + this.name + '.');
};

   obj = new Object();
person1[obj]
```

Please note that all keys in the square bracket notation are converted to String type, since objects in JavaScript can only have String type as key type. For example, in the above code, when the key obj is added to the person1, JavaScript will call the obj.toString()method, and use this result string as the new key.

Enumerate the properties of an object

Starting with ECMAScript 5, there are three native ways to list/traverse object properties:

- for...in loops
  This method traverses all enumerable properties of an object and its prototype chain

- Object.keys(o)

This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object o.

- Object.getOwnPropertyNames(o)

  This method returns an array containing all own properties' names (enumerable or not) of an object o.

You can also pass an object literal to the Object() constructor as a parameter, to prefill it with properties/methods. Try this:

```
var person1 = new Object({
  name: 'Chris',
  age: 38,
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
});
```

Object creation using the create() method:

However, some people prefer to create object instances without first creating constructors, especially if they are creating only a few instances of an object. JavaScript has a built-in method called create() that allows you to do that. With it, you can create a new object based on any existing object.

```
var person2 = Object.create(person1);
```

```
person2.name // ‘Chris’
person2.age // 38
person2.greeting()
```

You'll see that person2 has been created based on person1—it has the same properties and method available to it.

Prototype-type based language:

JavaScript is often described as a **prototype-based language** — each object has a **prototype object**, which acts as a template object that it inherits methods and properties from. An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is often referred to as a **prototype chain**, and explains why different objects have properties and methods defined on other objects available to them.

Well, to be exact, the properties and methods are defined on the prototype property on the Objects' constructor functions, not the object instances themselves.

In classic OOP, classes are defined, then when object instances are created all the properties and methods defined on the class are copied over to the instance. In JavaScript, they are not copied over — instead, a link is made between the object instance and its prototype (its __proto__ property, which is derived from the prototype property on the constructor), and the properties and methods are found by walking up the chain of prototypes

Understanding prototype objects:’

Let’s create a constructor function like

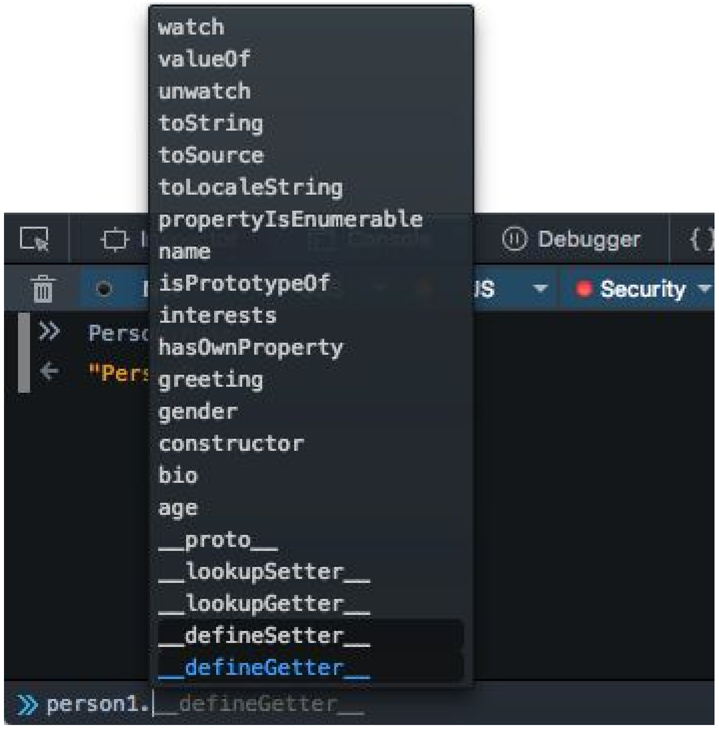
```
function Person(first, last, age, gender, interests) {

  // property and method definitions
```

}

We have then created object instance like this,

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

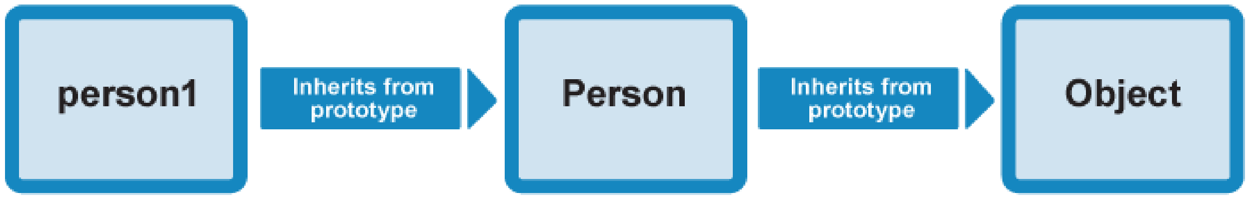
If you type "person1." into your JavaScript console, you should see the browser try to auto-complete this with the member available on this object:



```
watch
valueOf
unwatch
toString
toSource
toLocaleString
propertyIsEnumerable
name
isPrototypeOf
interests
hasOwnProperty
greeting
gender
constructor
bio
age
__proto__
__lookupSetter__
__lookupGetter__
__defineSetter__
__defineGetter__
```

```
>> person1.__defineGetter__
```

In this list, you will see the members defined on person1's prototype object, which is the Person()constructor — name, age, gender, interests, bio, and greeting. You will however also see some other members — watch, valueOf, etc — these are defined on the Person() constructor's prototype object, which is Object. This demonstrates the prototype chain working.



So what happens if you call a method on person1, which is actually defined on Object? For example:

```
person1.valueOf()
```

This method simply returns the value of the object it is called on — try it and see! In this case, what happens is:

- The browser initially checks to see if the person1 object has a valueOf() method available on it.
- It doesn't, so the browser then checks to see if the person1 object's prototype object (Person() constructor's prototype) has a valueOf() method available on it.
- It doesn't either, so the browser then checks to see if the Person() constructor's prototype object's prototype object (Object() constructor's prototype) has a valueOf() method available on it. It does, so it is called, and all is good!

Prototype property: where inherited members are defined

So, where are the inherited properties and methods defined? If you look at the Object reference page, you'll see listed in the left hand side a large number of properties and methods — many more than the number of inherited members we saw available on the person1 object in the above screenshot. Some are inherited, and some aren't — why is this?

The answer is that the inherited ones are the ones defined on the prototype property (you could call it a sub-namespace) — that is, the ones that begin with Object.prototype., and not the ones that begin with just Object. The prototype property's value is an object, which is basically a bucket for storing properties and methods that we want to be inherited by objects further down the prototype chain.

So Object.prototype.watch(), Object.prototype.valueOf(), etc., are available to any object types that inherit from Object.prototype, including new object instances created from the constructor.

Revisiting create():

Earlier on we showed how the Object.create() method can be used to create a new object instance.

1. For example, try this in your previous example's JavaScript console:
2. `var person2 = Object.create(person1);`
3. What create() actually does is to create a new object from a specified prototype object. Here, person2 is being created using person1 as a prototype object. You can check this by entering the following in the console:
4. `person2.__proto__`

This will return the person1 object.

The constructor property:

```
person1.constructor
```

These should both return the Person() constructor, as it contains the original definition of these instances.

2. A clever trick is that you can put parentheses onto the end of the constructor property (containing any required parameters) to create another object instance from that constructor.

```
var person3 = new person1.constructor('Karen', 'Stephenson', 26, 'female', ['playing drums', 'mountain climbing']);
```

3. Now try accessing your new object's features, for example:

```
person3.name.first
person3.age
person3.bio()
```

Modifying prototype:

1. Let's have a look at an example of modifying the prototype property of a constructor function (methods added to the prototype are then available on all object instances created from the constructor).

```
Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
};
```

2. Save the code and load the page in the browser, and try entering the following into the text input:

```
person1.farewell();
```

You should get an alert message displayed, featuring the person's name as defined inside the constructor. This is really useful, but what is even more useful is that the whole inheritance chain has updated dynamically, automatically making this new method available on all object instances derived from the constructor.

Defining properties inside the constructor & methods in a prototype:

You will rarely see properties defined on the `prototype` property, because they are not very flexible when defined like this. For example you could add a property like so:

```
Person.prototype.fullName = 'Bob Smith';
```

But this isn't very flexible, as the person might not be called that. It'd be much better to do this, to build the `fullName` out of `name.first` and `name.last`:

```
Person.prototype.fullName = this.name.first + ' ' + this.name.last;
```

This however doesn't work, as `this` will be referencing the global scope in this case, not the function scope. Calling this property would return `undefined undefined`. This worked fine on the method we defined earlier in the prototype because it is sitting inside a function scope, which will be transferred successfully to the object instance scope.

In fact, a fairly common pattern for more object definitions is to define the properties inside the constructor, and the methods on the prototype. This makes the code easier to read, as the constructor only contains the property definitions, and the methods are split off into separate blocks. For example:

```
// Constructor with property definitions

function Test(a, b, c, d) {
  // property definitions
};

// First method definition

Test.prototype.x = function() { ... }

// Second method definition

Test.prototype.y = function() { ... }

// etc.
```

Comparing Objects:

In JavaScript objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

Ex:1

```
// Two variables, two distinct objects with the same properties
var fruit = {name: 'apple'};
var fruitbear = {name: 'apple'};

fruit == fruitbear; // return false
fruit === fruitbear; // return false
```

```
// Two variables, a single object
var fruit = {name: 'apple'};
var fruitbear = fruit;  // assign fruit object reference to fruitbear

// here fruit and fruitbear are pointing to same object
fruit == fruitbear; // return true
fruit === fruitbear; // return true


fruit.name = 'grape';
console.log(fruitbear);    // yields { name: "grape" } instead of { name: "apple" }
```

Inheritance in Javascript:

In "classic OO" languages, you tend to define class objects of some kind, and you can then simply define which classes inherit from which other classes (see C++ inheritance for some simple examples). JavaScript uses a different system — "inheriting" objects do not have functionality copied over to them, instead the functionality they inherit is linked to via the prototype chain (often referred to as **prototypal inheritance**).

Let's explore how to do this with a concrete example (here, we have defined only the properties inside the constructor).

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
};
```

The methods are *all* defined on the constructor's prototype. For example:

```
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};
```

Say we wanted to create a Teacher class, like the one we described in our initial object-oriented definition, which inherits all the members from Person, but also includes:

1. A new property, subject — this will contain the subject the teacher teaches.
2. An updated greeting() method,

```
function Teacher(first, last, age, gender, interests, subject) {
  Person.call(this, first, last, age, gender, interests);

  this.subject = subject;
}
```

the `call()` function. This function basically allows you to call a function defined somewhere else, but in the current context. The first parameter specifies the value of `this` that you want to use when running the function, and the other parameters are those that should be passed to the function when it is invoked.

The last line inside the constructor simply defines the new `subject` property that teachers are going to have, which generic people don't have.

Inheriting from a constructor with no parameter:

```
function Brick() {
  this.width = 10;
  this.height = 20;
}
```

You could inherit the width and height properties by doing this

```
function BlueGlassBrick() {
  Brick.call(this);

  this.opacity = 0.5;
  this.color = 'blue';
}
```

Note that we've only specified this inside call() — no other parameters are required as we are not inheriting any properties from the parent that are set via parameters.

Setting teacher’s prototype and constructor reference:

All is good so far, but we have a problem. We have defined a new constructor, and it has a prototype property, which by default just contains a reference to the constructor function itself. It does not contain the methods of the Person constructor's prototype property. To see this, enter Object.getOwnPropertyNames(Teacher.prototype) into either the text input field or your JavaScript console. Then enter it again, replacing Teacher with Person. Nor does the new constructor *inherit* those methods. To see this, compare the outputs of Person.prototype.greeting and Teacher.prototype.greeting. We need to get Teacher() to inherit the methods defined on Person()'s prototype. So how do we do that?

```
Teacher.prototype = Object.create(Person.prototype);
```

Here our friend create() comes to the rescue again. In this case we are using it to create a new object and make it the value of Teacher.prototype. The new object has Person.prototype as its prototype and will therefore inherit, if and when needed, all the methods available on Person.prototype

2. We need to do one more thing before we move on. After adding the last line, Teacher.prototype's constructor property is now equal to Person(), because we just set Teacher.prototype to reference an object that inherits its properties from Person.prototype! Try saving your code, loading the page in a browser, and entering Teacher.prototype.constructor into the console to verify.

3. This can become a problem, so we need to set this right. You can do so by going back to your source code and adding the following line at the bottom:

```
Teacher.prototype.constructor = Teacher;
```

4. Now if you save and refresh, entering Teacher.prototype.constructor should return Teacher(), as desired, plus we are now inheriting from Person()!

Giving teacher’s new greeting functions:

```
Teacher.prototype.greeting = function() {
  var prefix;

  if (this.gender === 'male' || this.gender === 'Male' || this.gender === 'm' || this.gender
=== 'M') {
    prefix = 'Mr.';
  } else if (this.gender === 'female' || this.gender === 'Female' || this.gender === 'f' ||
this.gender === 'F') {
    prefix = 'Mrs.';
  } else {
    prefix = 'Mx.';
  }

  alert('Hello. My name is ' + prefix + ' ' + this.name.last + ', and I teach ' + this.subject +
'.');
};
```

```
teacher1.name.first;
teacher1.interests[0];
teacher1.bio();
teacher1.subject;
teacher1.greeting()
```

Object member summary:

To summarize, you've basically got three types of property/method to worry about:

1. Those defined inside a constructor function that are given to object instances. These are fairly easy to spot — in your own custom code, they are the members defined inside a constructor using the this.x = x type lines; in built in browser code, they are the members only available to object instances (usually created by calling a constructor using the new keyword, e.g. var myInstance = new myConstructor()).
2. Those defined directly on the constructor themselves, that are available only on the constructor. These are commonly only available on built-in browser objects,

and are recognized by being chained directly onto a constructor, *not* an instance. For example, Object.keys().

3. Those defined on a constructor's prototype, which are inherited by all instances and inheriting object classes. These include any member defined on a Constructor's prototype property, e.g. myConstructor.prototype.x().

Proxy:

The Proxy object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)

Ex 1:

```
let bears = { grizzly : true }
let grizzlyCount =0
const proxyBears = new Proxy( bears, {
       get: function (target, prop) {
           if (prop === grizzly) { grizzlyCount ++ }
           return target[prop]
      }
      set: function (target, prop, value) {
           if ( [‘grizzly’, ‘brown’, ‘polar’].indexOf(prop) === -1 )
           {
               throw new Error (‘That is totally not a bear !!’)
           }
           target[prop] = value
      }
)

proxyBears.grizzly // true
proxyBears.grizzly // true
proxyBears.grizzly // true
proxyBears.grizzly // true
console.log( grizzlyCount) // 4
```

```
proxyBears.adsfsfsd = true // Uncaught Error : That is totally not a bear !!
proxyBears.polar = true
console.log(proxyBears.polar) // true

delete proxyBears.polar
```

Ex 2:

```
function growl() {
   return ‘grrr’;
}

const loudGrowl = new Proxy(growl, {
    apply: function (target, thisArg, args) {
      return target().toUpperCase() + ‘!!!!’
   }
})

console.log(loudGrowl()) // ‘GRRR!!!!’
```

Ex: 3

```
const person = {
first : ‘Gautham’,
last: ‘Sundar’
}
const cleverPerson = new Proxy (person, {
    get: function (target, prop) {
        if(! (prop in target) ) {
            return prop.split(‘_’).map( function (part) {
               return target[part]
            }).join(‘ ‘)
        }
       return target[prop]
   }
})

console.log(cleverPerson.first_last)
```

Accessing Json:

We have made our JSON data available on our GitHub, at https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json.

To obtain the JSON, we are going to use an API called XMLHttpRequest (often called **XHR**). This is a very useful JavaScript object that allows us to make network requests to retrieve resources from a server via JavaScript (e.g. images, text, JSON, even HTML snippets), meaning that we can update small sections of content without having to reload the entire page. This has led to more responsive web pages, and sounds exciting, but it is unfortunately beyond the scope of this article to teach it in much more detail.

To start with, we are going to store the URL of the JSON we want to retrieve in a variable. Add the following at the bottom of your JavaScript code:

```
var requestURL =
'https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json';
```

To create a request, we need to create a new request object instance from the XMLHttpRequest constructor, using the newkeyword. Add the following below your last line:

```
var request = new XMLHttpRequest();
```

Now we need to open a new request using the open() method. Add the following line:`request.open('GET', requestURL);`

This takes at least two parameters — there are other optional parameters available. We only need the two mandatory ones for this simple example:

- The HTTP method to use when making the network request. In this case GET is fine, as we are just retrieving some simple data.
- The URL to make the request to — this is the URL of the JSON file that we stored earlier.

Next, add the following two lines — here we are setting the responseType to JSON, so that XHR knows that the server will be returning JSON, and that this should be converted behind the scenes into a JavaScript object. Then we send the request with the send() method:

```
request.responseType = 'json';
request.send();
```

The last bit of this section involves waiting for the response to return from the server, then dealing with it. Add the following code below your previous code

```
request.onload = function() {
  var superHeroes = request.response;
  populateHeader(superHeroes);
  showHeroes(superHeroes);
}
```

Here we are storing the response to our request (available in the response property) in a variable called superHeroes; this variable will now contain the JavaScript object based on the JSON! We are then passing that object to two function calls — the first one will fill the <header> with the correct data, while the second one will create an information card for each hero on the team, and insert it into the <section>.

We have wrapped the code in an event handler that runs when the load event fires on the request object (see onload) — this is because the load event fires when the response has successfully returned; doing it this way guarantees that request.responsewill definitely be available when we come to try to do something with it.

```
function populateHeader(jsonObj) {
  var myH1 = document.createElement('h1');
  myH1.textContent = jsonObj['squadName'];
  header.appendChild(myH1);

  var myPara = document.createElement('p');
  myPara.textContent = 'Hometown: ' + jsonObj['homeTown'] + ' // Formed: ' +
jsonObj['formed'];
  header.appendChild(myPara);
}

function showHeroes(jsonObj) {
  var heroes = jsonObj['members'];

  for (var i = 0; i < heroes.length; i++) {
    var myArticle = document.createElement('article');
    var myH2 = document.createElement('h2');
    var myPara1 = document.createElement('p');
    var myPara2 = document.createElement('p');
    var myPara3 = document.createElement('p');
    var myList = document.createElement('ul');

    myH2.textContent = heroes[i].name;
    myPara1.textContent = 'Secret identity: ' + heroes[i].secretIdentity;
    myPara2.textContent = 'Age: ' + heroes[i].age;
    myPara3.textContent = 'Superpowers:';

    var superPowers = heroes[i].powers;
    for (var j = 0; j < superPowers.length; j++) {
      var listItem = document.createElement('li');
      listItem.textContent = superPowers[j];
      myList.appendChild(listItem);
    }

    myArticle.appendChild(myH2);
    myArticle.appendChild(myPara1);
    myArticle.appendChild(myPara2);
    myArticle.appendChild(myPara3);
    myArticle.appendChild(myList);

    section.appendChild(myArticle);
  }
}
```

Parse vs stringify:

- stringify(): Accepts an object as a parameter, and returns the equivalent JSON string form.

```
console.log(JSON.stringify({ x: 5, y: 6 }));
// expected output: "{"x":5,"y":6}"
```

- parse(): Accepts a JSON string as a parameter, and returns the corresponding JavaScript object.

```
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"foo"');           // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('null');            // null
```

In continuation
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_building_practice

----

Console:

The console is a tool that developers use to record the output of their JavaScript programs.
The `console.log()` command is used to print, or log, text to the console.

Data types:

Data types are the building blocks of all languages and essential pieces of any program.

Below are examples of four *primitive data types* that lay the foundation for all JavaScript programs. Primitive data types, as their name implies, are the simplest built-in forms of data.

The data types include:

1. Strings (console.log(“Java Script”)
2. Numbers (console.log(33.7))
3. Booleans (console.log(true))
4. Null (console.log(true))

Properties:

When you introduce a new piece of data into a JavaScript program, the browser saves it as an instance of the data type. An *instance* is an individual case (or object) of a data type.

JavaScript will save a new piece of data, like `'Hello'`, as a string *instance* in the computer's memory. Another example, the number `40.7`, is stored as an instance of the number data type.

An instance, like the string `'Hello'`, has additional information attached to it.

For example, every string instance has a property called `length` that stores the number of characters in it. You can retrieve property information by appending the string with a period and the property name:

```
console.log(‘Hello’.length)
```

Built-in methods:

While the length of a string is calculated when an instance is created, a string instance also has *methods* that calculate new information as needed. When these built-in methods are called on an instance, they perform actions that generate an output.

Built-in methods are called, or used, by appending an instance with a period, the name of the method, and opening (() and closing ()) parentheses. Consider the examples below:

```
console.log('Hello'.toUppercase() );
console.log('        Remove Whitespace      '.trim() )
```

Libraries:

Instance methods, by definition, require that you create an instance before you can use them. What if you want to call a method without an instance? That's where JavaScript libraries come in. Libraries contain methods that you can call without creating an instance.

One such collection contains mathematical methods, aptly named the `Math` library.

Let's see how you call the `.random()` method from the `Math` library:

```
console.log(Math.random() );
```

Global scope :

Variables defined in the global scope are declared outside of a set of curly braces `{}`, referred to as a *block*, and are thus available throughout a program.

```
const color = 'blue'

const colorOfSky = () => {
  return color;
};

console.log(colorOfSky());
```

Here the variable color is declared *outside* of the function block, giving it global scope.

In turn, color can be accessed within the colorOfSky function.

const vs let :

Variables declared with `let` can be reassigned.

Variables that are assigned with `const` cannot be reassigned. However, arrays that are declared with `const` remain *mutable*, or changeable.

This means that we can change the contents of an array, but cannot reassign the variable name to a new array or other data type.

```
let condiments = ['Ketchup', 'Mustard', 'Soy Sauce', 'Sriracha'];
condiments.push("gautham")
condiments= ['a string'];
console.log(condiments)

const utensils = ['Fork', 'Knife', 'Chopsticks', 'Spork'];
utensils.push("gautham")
utensils.pop();
utensils[2]="sundar" // no error,  since it just changes the value
utensils=["sundar"] // throws an error
```

This means that, even when declared with **const, arrays are still mutable**; they can be changed. However, a variable declared with `const` cannot be reassigned.

However, the properties of objects assigned to constants are not protected, so the following statement is executed without problems.

```
const MY_OBJECT = {'key': 'value'};
MY_OBJECT.key = 'otherValue';
```

.forEach()

`.forEach()` will execute the same code on each element of an array.

```
let groceries = ['whole wheat flour', 'brown sugar', 'salt', 'cranberries', 'walnuts'];

groceries.forEach(function(groceryItem) {
  console.log(' - ' + groceryItem);
});
```

1. `groceries.forEach` calls the `.forEach()` method on the groceries array.
2. `(function(groceryItem) {` creates a function that takes a single parameter, `groceryItem` and opens the block of code for that function. Because `.forEach()` is an iterator method, every element in the `groceries` array will be passed to this function as an argument in place of `groceryItem`
3. `});` closes the function code block and `.forEach()` method in that order.

If using Array syntax method,

```
groceries.forEach( (groceryItem) => { console.log(' - ' + groceryItem) } );
```

.map() vs .forEach()

```
let numbers = [1, 2, 3, 4, 5];
```

Ex:1

```
let bigNumbers = numbers.map(function(number) {
  return number * 10;
});

console.log( bigNumbers) // [ 10,20,30,40,50 ]
```

Ex:2 :

```
let bigNumbers =[]
numbers.forEach(function(number,i) {
  bigNumbers[i] = number * 10;
});

console.log( bigNumbers) // [ 10,20,30,40,50 ]
```

.map()

In previous exercise, we called .forEach() method and learned that it returns `undefined`. It also does not change the array it is called upon. What if we do want to change the contents of the array? We can use `.map()`

```
let numbers = [1, 2, 3, 4, 5];

let bigNumbers = numbers.map(function(number) {
  return number * 10;
});
```

1. The first line is an a `let bigNumbers = numbers.map` creates a new array, `bigNumbers`, in which the returned values of the `.map()` method will be saved and calls the `.map()` method on the `numbers` array.
2. `(function(number) {` creates a function that takes a single parameter, `number`, and opens the block of code for that function.
3. `return number * 10;` is the code we wish to execute upon each element in the array. This will save each value from the `numbers` array, multiplied by `10`, to the `bigNumbers` array.
4. `});` closes the function code block and `.map()` method in that order.

If using Arrow syntax method,

```
let numbers = [1, 2, 3, 4, 5];
let bigNumbers = numbers.map( (numbers) => { numbers * 10 } );
```

.filter()

Another useful iterator method is `.filter()`. Like `.map()`, `.filter()` returns a new array. However, `.filter()` returns certain elements from the original array that evaluate to truthy based on conditions written in the block of the method.

```
let words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];
let shortWords = words.filter(function(word) {
  return word.length < 6;
});

// shortWords has [‘chair’ , ‘music’,’brick’, ‘pen’, ‘door’]
```

If using arrow syntax method,

```
let shortWords = words.filter(word => word.length < 6);
```

.some() , .every()

.some() and .every() return a boolean value, based on the condition specified.

```
let words = ['unique', 'uncanny', 'pique', 'oxymoron', 'guise'];
console.log(words.some(function(word) {
  return word.length < 6;
}));

let interestingWords = words.filter(word => word.length > 5);
console.log(interestingWords.every(word =>  word.length > 6      ));
```

Iteration Documentation:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

----

Class:

```
class Surgeon {
  constructor(name, department) {
    this._name = name;
    this._department = department;
    this._remainingVacationDays = 20;
  }
  get name() {
    return this._name
  }
  get  department() {
    return this._department
  }
  get remainingVacationDays(){
    return this._remainingVacationDays
  }
  takeVacationDays(daysoff) {
    this._remainingVacationDays = this._remainingVacationDays - daysoff;
  }
}

const surgeonCurry = new Surgeon('Curry', 'Cardiovascular');
const surgeonDurant = new Surgeon('Durant', 'Orthopedics');

console.log(surgeonCurry.name)
surgeonCurry.takeVacationDays(3)
console.log(surgeonCurry.remainingVacationDays)
```

Inheritance :

```
class HospitalEmployee {
  constructor(name) {
    this._name = name;
    this._remainingVacationDays = 20;
  }

  get name() {
    return this._name;
  }

  get remainingVacationDays() {
    return this._remainingVacationDays;
  }

  takeVacationDays(daysOff) {
    this._remainingVacationDays -= daysOff;
  }

  static generatePassword() {
    return Math.floor(Math.random() * 10000)
  }
}

class Nurse extends HospitalEmployee {
  constructor(name, certifications) {
    super(name);
    this._certifications = certifications;
  }

  get certifications() {
    return this._certifications;
  }

  addCertification(newCertification) {
    this.certifications.push(newCertification);
  }
}

const nurseOlynyk = new Nurse('Olynyk', ['Trauma','Pediatrics']);
nurseOlynyk.takeVacationDays(5);
console.log(nurseOlynyk.remainingVacationDays);
nurseOlynyk.addCertification('Genetics');
console.log(nurseOlynyk.certifications);
```

In the example above, we create a new class named Nurse that extends the HospitalEmployee class. Let's pay special attention to our new keywords: extends and super.

- The extends keyword makes the methods of the HospitalEmployee class available inside the Nurse class.
- The super keyword calls the constructor of the parent class. In this case, super(name)passes the name argument of the Nurse class to the constructor of the HospitalEmployee class. When the HospitalEmployee constructor runs, it sets this._name = name; for new Nurse instances.

Notice, we call super on the first line of our constructor(), then set the certifications property on the second line. In a constructor(), you must always call the super method before you can use the this keyword — if you do not, JavaScript will throw a reference error. To avoid reference errors, it is best practice to call superon the first line of subclass constructors.

Static methods:

Sometimes you will want a class to have methods that aren't available in individual instances, but that you can call directly from the class.

Take the Date class, for example — you can both create Date instances to represent whatever date you want, and call *static* methods, like Date.now() which returns the current date, directly from the class. The .now() method is static, so you can call it directly from the class, but not from an instance of the class.

Browser compatibility and transpilation:

You're probably prompted to update your web browser every few months. Do you know why? A few reasons include addressing security vulnerabilities, adding features, and supporting new HTML, CSS, and JavaScript syntax.

The reasons above imply there is a period before a software update is released when there are security vulnerabilities and unsupported language syntax.

This lesson focuses on the latter. Specifically, how developers address the gap between the new JavaScript syntax that they use, and the JavaScript syntax that web browsers recognize.

This has become a widespread concern for web developers since Ecma International, the organization responsible for standardizing JavaScript, released a new version of it in 2015, called ECMA Script2015, commonly referred to as ES6. Note, the 6 refers to the version of JavaScript and is not related to the year it was released (the previous version was ES5).

Upon release, web developers quickly adopted the new ES6 syntax, as it improved readability and efficiency. However, ES6 was not supported by most web browsers, so developers ran into browser compatibility issues.

In this lesson, you will learn about two important tools for addressing browser compatibility issues.

- caniuse.com — A website that provides data on web browser compatibility for HTML, CSS, and JavaScript features. You will learn how to use it to look up ES6 feature support.
- Babel — A Javascript library that you can use to convert new, unsupported JavaScript (ES6), into an older version (ES5) that is recognized by most modern browsers.

Why ES6

The version of JavaScript that preceded ES6 is called JavaScript ES5. Three reasons for the ES5 to ES6 updates are listed below:

- A similarity to other programming languages — JavaScript ES6 is syntactically more similar to other object-oriented programming languages. This leads to less friction when experienced, non-JavaScript developers want to learn JavaScript.
- Readability and economy of code — The new syntax is often easier to understand (more readable) and requires fewer characters to create the same functionality (economy of code).
- Addresses sources of ES5 bugs — Some ES5 syntax led to common bugs. With ES6, Ecma introduced syntax that mitigates some of the most common pitfalls.

To limit the impact of ES6 browser compatibility issues, Ecma made the new syntax backwards compatible, which means you can map JavaScript ES6 code to ES5.

New features in ES6 :

1. Destructuring assignment
2. Maps
3. Sets
4. Arrow function
5. Default and Rest parameter
6. Spread operator
7. Default & Named export , import
8. Let and const
9. Iterators & Iterables

10. Proxies

Changing from ES6 syntax to ES5:

The `let` and `const` keywords were introduced in ES6. Before that, we declared all variables with the `var` keyword.

```
var pasta = "Spaghetti"; // ES5 syntax

const meat = "Pancetta"; // ES6 syntax

let sauce = "Eggs and cheese"; // ES6 syntax

// Template literals, like the one below, were introduced in ES6
const carbonara = `You can make carbonara with ${pasta}, ${meat}, and a sauce made with
${sauce}.`;
```

is changed to ES 5

```
var pasta = "Spaghetti";

var meat = "Pancetta";

var sauce = "Eggs and cheese";

var carbonara = 'You can make carbonara with' + pasta +',' meat +', and a sauce made with' +
sauce.;
```

Transpilation:

So far, we manually converted ES6 code to ES5. Although manual conversion only took you a few minutes, it is unsustainable as the size of the JavaScript file increases.

Because ES6 is predictably backwards compatible, a collection of JavaScript programmers developed a JavaScript library called **Babel that *transpiles* ES6 JavaScript to ES5.**

Transpilation is the process of converting one programming language to another.

We will pass JavaScript ES6 code to Babel, which will transpile it to ES5 and write it to a file in the **lib** directory. To install babel,

```
npm install babel-cli
```

```
npm install babel-preset-env
```

and if we type `npm run build`, we can view the ES5 code in **./lib/main.js**.

Module export:

We can get started with modules by defining a module in one file, and making the module available for use in another file. Below is an example of how to define a module, and how to export it .

The pattern we use to export modules is thus:

1. Define an object to represent the module.
2. Add data or behavior to the module.
3. Export the module.

```
let Menu = {};
Menu.specialty = "Roasted Beet Burger with Mint Sauce";

module.exports = Menu;
```

Let's consider what this code means.

1. `module.exports = Menu;` exports the `Menu` object as a module. `module` is a variable that represents the module, and `exports` exposes the module as an object.

require() :

To make use of the exported module and the behavior we define within it, we import the module. A common way to do this is with the require() function.

We can create a separate file **order.js** and import the Menu module from **menu.js** to **order.js** using require():

In **order.js** we would write:

```
const Menu = require('./menu.js');

function placeOrder() {
  console.log('My order is: ' + Menu.specialty);
}

placeOrder();
```

We now have the entire behavior of Menu available in **order.js**. Here, we notice:

1. In **order.js** we import the module by creating a variable with const called Menu and setting it equal to the value of the require() function. We can set this variable to any variable name we like, such as menuItems.
2. require() is a JavaScript function that enables a module to load by passing in the module's filepath as a parameter.
3. './menu.js' is the argument we pass to the require() function.
4. The placeOrder() function then uses the Menu module in its function definition. By calling Menu.specialty, we access the property specialty defined in the Menu module.
5. We can then invoke the function using placeOrder()

We can also wrap any collection of data and functions in an object, and export the object using module.exports. Consider the below code,

File : ‘2-airplane.js’

```
let Airplane = {}

module.exports = {
  myAirplane: "CloudJet",
  displayAirplane: function() {
	return this.myAirplane;
}
};
```

In the code above, `module.exports` exposes the current module as an object.

File: ’2-missionControl.js’

```
const Airplane = require('./2-airplane.js')
console.log(Airplane.displayAirplane())
```

----

export default:

As of ES6, JavaScript has implemented a new more readable and flexible syntax for exporting modules. These are usually broken down into one of two techniques, ***default export*** and ***named exports***.

We'll begin with the first syntax, default export. The default export syntax works similarly to the `module.exports` syntax, allowing us to export one module per file.

Let's look at an example.

```
let Airplane = {
}

Airplane.availableAirplanes = [
  {
    name: "AeroJet",
    fuelCapacity: 800
  },
  {
    name: "SkyJet",
    fuelCapacity: 500
  }
]

export default Airplane;
```

import

ES6 module syntax also introduces the `import` keyword for importing objects.

```
import Airplane from './airplane';

let displayFuelCapacity = () => {
  Airplane.availableAirplanes.forEach(
  function(element) {
    console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
  }
  );
}

displayFuelCapacity();
```

Named export:

ES6 introduced a second common approach to export modules. In addition to default export, *named exports* allow us to export data through the use of variables.

```
let availableAirplanes = [
{
 name: 'AeroJet',
 fuelCapacity: 800,
 availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],
},
{
name: 'SkyJet',
 fuelCapacity: 500,
 availableStaff: ['pilots', 'flightAttendants']
}
];

let flightRequirements = {
  requiredStaff: 4,
};

function meetsStaffRequirements(availableStaff, requiredStaff) {
  if (availableStaff.length >= requiredStaff) {
    return true;
  } else {
    return false;
  }
};

export { availableAirplanes, flightRequirements, meetsStaffRequirements};
```

Named import:

To import objects stored in a variable, we use the `import` keyword and include the variables in a set of `{}`.

```
import {availableAirplanes, flightRequirements, meetsStaffRequirements} from './airplane';

function displayFuelCapacity() {
  availableAirplanes.forEach(function(element) {
    console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
  });
}

displayFuelCapacity();
```

```
function displayStaffStatus() {
  availableAirplanes.forEach(function(element) {
   console.log(element.name + ' meets staff requirements: ' +
meetsStaffRequirements(element.availableStaff, flightRequirements.requiredStaff) );
  });
}

displayStaffStatus();
```

Named exports are also distinct in that they can be exported as soon as they are declared, by placing the keyword `export` in front of variable declarations.

```
export let availableAirplanes = [{
 name: 'AeroJet',
 fuelCapacity: 800,
 maxSpeed:1200,
 minSpeed: 300,
 availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],
},
{name: 'SkyJet',
 fuelCapacity: 500,
 maxSpeed: 800,
 minSpeed: 200,
 availableStaff: ['pilots', 'flightAttendants']
}];

export let flightRequirements = {
  requiredStaff: 4,
  requiredSpeedRange:700
};

export function meetsStaffRequirements(availableStaff, requiredStaff) {
  if (availableStaff.length >= requiredStaff) {
    return true;
  } else {
    return false;
  }
};
```

```
export function meetsSpeedRangeRequirements(maxSpeed,minSpeed,requiredSpeedRange) {
  let range= maxSpeed- minSpeed;
  if(range>requiredSpeedRange) {
    return true;
  }
  else {
    return false;
  }
}
```

----

export as :

Named exports also conveniently offer a way to change the name of variables when we export or import them.

```
let availableAirplanes = [
{name: 'AeroJet',
 fuelCapacity: 800,
 availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],
 maxSpeed: 1200,
 minSpeed: 300
},
{name: 'SkyJet',
 fuelCapacity: 500,
 availableStaff: ['pilots', 'flightAttendants'],
 maxSpeed: 800,
 minSpeed: 200
}
];

let flightRequirements = {
  requiredStaff: 4,
  requiredSpeedRange: 700
};

function meetsStaffRequirements(availableStaff, requiredStaff) {
  if (availableStaff.length > requiredStaff) {
    return true;
  } else {
    return false;
  }
};

function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange) {
  let range = maxSpeed - minSpeed;
  if (range > requiredSpeedRange) {
    return true;
    } else {
    return false;
  }
};

export {availableAirplanes as aircrafts, flightRequirements as flightReqs,
meetsStaffRequirements as meetsStaffReqs, meetsSpeedRangeRequirements as
meetsSpeedRangeReqs };
```

Import as

To import named export aliases with the `as` keyword, we add the aliased variable in our import statement.

1. One way is to import entire module as alias

```
import * as Carte from './menu';

Carte.availableAirplanes;
Carte.meetsStaffRequirements;
Carte.flightRequirements;
```

2. Another way is to change each variable to its alias, with the exception of the variable in the import statement.

```
import {availableAirplanes as aircrafts, flightRequirements as flightReqs,
meetsStaffRequirements as meetsStaffReqs, meetsSpeedRangeRequirements as
meetsSpeedRangeReqs} from './airplane';


function displayFuelCapacity() {
  aircrafts.forEach(function(element) {
    console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
  });
}

displayFuelCapacity();

function displayStaffStatus() {
  aircrafts.forEach(function(element) {
   console.log(element.name + ' meets staff requirements: ' +
meetsStaffReqs(element.availableStaff, flightReqs.requiredStaff) );
  });
}

displayStaffStatus();

function displaySpeedRangeStatus() {
  aircrafts.forEach(function(element) {
   console.log(element.name + ' meets speed range requirements:' +
meetsSpeedReqs(element.maxSpeed, element.minSpeed, flightReqs.requiredSpeedRange));
  });
}

displaySpeedRangeStatus();
```

Combining Export statements:

We can also use named exports and default exports together.

```
let specialty = '';
function isVegetarian() {
};
```

```
function isLowSodium() {
};
function isGlutenFree() {
};

export { specialty as chefsSpecial, isVegetarian as isVeg };

export default isGlutenFree;
```

Here we use the keyword `export` to export the named exports at the bottom of the file. Meanwhile, we export the `isGlutenFree` variable using the `export default` syntax.

This would also work if we exported most of the variables as declared and exported others with the `export default` syntax.

```
export let Menu = {};

export let specialty = '';

export let isVegetarian = function() {
};

export let isLowSodium = function() {
};

let isGlutenFree = function() {
};

export default isGlutenFree;
```

While it's better to avoid combining two methods of exporting, it is useful on occasion. For example, if you suspect developers may only be interested in importing a specific function and won't need to import the entire default export.

Combining Import statements:

We can import the collection of objects and functions with the same data.

We can use an `import` keyword to import both types of variables as such:

```
import {availableAirplanes,flightRequirements,meetsStaffRequirements} from './airplane';
import meetsSpeedRangeRequirements from './airplane';
```

HTTP Requests:

AJAX involves requesting data over the web, which is done using HTTP Requests. There are four commonly used types of HTTP requests - `GET`, `POST`, `PUT`, and `DELETE`.

`GET` requests receive information from other sites by sending a *query*.
`POST` requests can change information on another site and will receive information or data in response.

There are several differences between the way `GET` and `POST` requests are made. A `GET` request is like a search. If you navigate to Google and search for something, you might notice that all of your search terms are added to the URL, like this: “ https://www.google.com/#q=pizza+near+union+square “
This URL is saying "Google, please retrieve a list of pizza near Union Square." It is not introducing any new information to Google.

`POST` requests, on the other hand, introduce new information to another website. Instead of sending this information in the URL of the request, it is sent as part of the body of the request.

XHR GET Requests:

When AJAX was first formalized by the World Wide Web Consortium in 2006, it required the use of an ***XMLHttpRequest*** *object*, a JavaScript object that is used to retrieve data.

There are several steps to creating an AJAX request using an XMLHttpRequest, or XHR, object.

```
// GET
const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';
xhr.responseType = 'json';
xhr.onreadystatechange = function() {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    console.log(xhr.response);
  }
};
xhr.open('GET', url);
xhr.send();
```

On the first line, we create an XMLHttpRequest object. We do that by typing new, then the type of object, XMLHttpRequest(). This is called the *new operator*.

The object is saved in a const called xhr. Throughout this request, we'll be accessing properties of this object.

On the next line, we save the URL to which we're going to make our request in a const called url.

We set the responseType of the xhr object to 'json'. There are other possible response types, which you can read about here.

onreadystatechange is an event handler that is called whenever the value of the readyState property changes. We set it equal to an anonymous function. This function will handle the response to our request.

First, we check if the readyState of our xhr object is equal to XMLHttpRequest.DONE. If that evaluates to true, the code in the block executes. It is useful while writing a new program to log the response to the console so that you can see its structure. Later, you might change this function to add parts of the response to your web page or do something else with it entirely.

Then, we call the .open() method on our xhr object and pass it two arguments - 'GET' (the type of request) and url, the URL we're querying. .open() creates and structures the request. It tells the request what method to use, GET or POST, and what URL to query.

Finally, we call the .send() method on our xhr object and pass it no arguments. This is because data sent in GET requests is sent as part of the URL. Calling the .send() method sends the xhr object with its relevant information to the API URL.

XHR POST Request:

In a GET request, the query information (what you're asking the server to return) is generally sent as part of the URL whereas in a POST request, the information is sent to the server as part of the body of the request. This information is created and saved in the data constant and sent to the API as an argument passed to the .send() method.

The object containing this data is passed to the JSON.stringify() method, which will format the object as a string. This is saved to a const called data.

```
const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';
const data = JSON.stringify({
  id: '200'
});
xhr.responseType = 'json';
xhr.onreadystatechange = function() {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    console.log(xhr.response);
  }
};
xhr.open('POST', url);
xhr.send(data);
```

jQuery:

The same requests that you've been writing by constructing XMLHttpRequest objects and using vanilla JavaScript to send the request and handle the response can be written much more simply using some methods from the jQuery library.

```
<script src='https://code.jquery.com/jquery-3.2.1.min.js'></script>
```

This script points to the jQuery CDN which allows us to use jQuery in our project. Because we were already using it to simplify accessing elements and updating the web page with new information, it makes sense to utilize its other functionalities as well.

$.ajax() is a method provided by the jQuery library specifically to handle AJAX requests. All parts of the request are included in a single object that is passed to the method as an argument.

```
$.ajax({
  url: 'https://api-to-call.com/endpoint',
  type : 'GET',
  dataType : 'json',
  success(response) {
                        console.log(response);
},
  error(jqXHR,status, errorThrown){
    console.log(jqXHR)
  }
});
```

1. The first property of the settings object is url and its value is the endpoint of the API.
2. The second property is the type of request, in this case, 'GET'. This property is optional for GET requests because it is the default setting.
3. The next property specifies the dataType, which is 'json' in this case.
4. Then, we include the success function which will handle the response if our request is successful. We create it with a single parameter, response. Inside the function's code block, we will specify what to do with the response object.
5. Finally, we include the error function which will handle the response if our request is not successful. This function is created with three parameters, jqXHR, status, and errorThrown. jqXHR is the XHR response object returned by the $.ajax() method. It will contain information about the error.

```
//post
$.ajax({
  url: 'https://api-to-call.com/endpoint',
  type :  'POST',
  data : JSON.stringify({id:200}),
  dataType: 'json',
  success(response) {
    console.log(response)
  },
  error(jqXHR,status,errorThrown) {
    console.log(jqXHR)
  }
});
```

$.get() :

The jQuery library provides other methods that allow us to write fewer lines of code to accomplish the same goals.

```
$.get('https://api-to-call.com/endpoint', response => {...}, 'json');
```

In the example above, we use the `$.get()` method to write a `GET` request. However, we've gone from nearly a dozen lines of code at the beginning of this lesson to one or two.

1. `$.get(` opens the method call.
2. `'https://api-to-call.com/endpoint'` is the URL to which we're making our request.
3. The second parameter, `response => {...}` is an arrow function. This is the success callback function. Between the curly braces, specify what to do with the response if it is successful, such as logging it to the console or appending it to the body of the HTML.
4. The third parameter, `'json'`, is the response format.

$.post():

Just like the `$.get()` method, jQuery provides a `$.post()` method to make writing AJAX POST requests simpler.

```
$.post('https://api-to-call.com/endpoint', {data}, response => {...}, 'json');
```

In the example above, we use the `.post()` method to write a POST request using AJAX. It should look very similar to what you saw in the previous exercise!

1. `$.post(` opens the method call.
2. `'https://api-to-call.com/endpoint'` is the URL to which we're making our request.

3. The second parameter, `{data}` is the object you'll use to send data with your request.
4. The third parameter, `response => {...}` is an arrow function. This is the success callback function. Between the curly braces, you would specify what you want to do with the response if it is successful, such as logging it to the console or appending it to the body of the HTML.
5. The fourth parameter, `'json'`, is the response format.

```
$.ajax({
  url: urlWithKey,
  data : JSON.stringify({longUrl: urlToShorten}),
  dataType : 'json',
  type : ‘post’,
  contentType : 'application/json',
  success(response) {
    $responseField.append('<p>Your shortened url is: </p><p>' + response.id + '</p>');
  },
  error(jqXHR,status,errorThrown) {
  console.log(jqXHR);
}
});
```

is equivalent to

```
$.post({
  url: urlWithKey,
  data : JSON.stringify({longUrl: urlToShorten}),
  dataType : 'json',
  contentType : 'application/json',
  success(response) {
    $responseField.append('<p>Your shortened url is: </p><p>' + response.id + '</p>');
  },
  error(jqXHR,status,errorThrown) {
  console.log(jqXHR);
}
});
```

$.getJSON()

There are other helper methods that can reduce the amount of boilerplate code necessary. If you know, for example, that you want your data type to be JSON, you can use the `$.getJSON()` method in place of the `$.get()` method.

```
$.get(urlToExpand,response => {
 $responseField.append('<p>Your expanded url is: </p><p>' +
 response.longUrl + '</p>');
},'json');
```

is equivalent to

```
$.getJSON(urlToExpand,response => {
 $responseField.append('<p>Your expanded url is: </p><p>' +
 response.longUrl + '</p>');
});
```

----

fetch() Get request:

```
fetch('https://api-to-call.com/endpoint').then(
response => {
        if (response.ok) {
                return response.json();
        }
  throw new Error('Request failed!');
}, networkError => {
  console.log(networkError.message);
}).then(jsonResponse => jsonResponse);
```

The fetch() function

1. creates a request object using the information provided to it.
2. sends that request object to the URL provided
3. returns a Promise that ultimately resolves to a response object, which contains a lot of information, including (if everything went well), the information requested.

- We chain a .then() method to the closing parentheses of the fetch() function. This is where the asynchronicity of JavaScript comes in - the fetch() function makes the request and returns the response, and we don't call the function that will handle the response until it has been received.
- .then() takes two callback functions as parameters, the first of which handles success and the second of which handles failure.
- The first callback function takes response as a parameter. response is the resolution of the Promise returned by the fetch() function.

On the next line, we use a conditional statement to check the ok property of the response object. If it evaluates to a truthy value, we call the .json() method on the response object. This method takes the information from the response and converts it to a JSON object whose properties and values we can access.

Because the .json() method takes some time to implement, it returns a Promise that will eventually resolve with the desired JSON object.

We will then chain another .then() method call to use the converted response object. We return response.json().

Outside of the code block of the conditional statement, we throw new Error('Request failed'). This error will only be thrown if response.json() is not returned because the response was not ok.

Then, we create the second callback function, which takes a single parameter, networkError. This function will only be called if there is a network error, such as a 500 error. In this function's code block, we log networkError.message to the console so that we can determine what went wrong. The .then() method is closed on the next line with a closing parenthesis ()).

- At the end of the first .then() method, we chain another .then() method that takes a single callback function as an argument. This callback function's parameter is jsonResponse and this function will handle the jsonResponse object. This is an object that contains the information we requested from the API and we can use that information on our website.

fetch() Post request:

In the previous exercise, you created the query URL, called the fetch() function and passed it the query URL and a settings object. Then, you chained a .then() method and passed it two functions as arguments - one to handle the Promise if it resolves, and one to handle network errors if the Promise is rejected.

```
fetch('https://api-to-call.com/endpoint',{ method: 'POST', body: JSON.stringify({id:'200'}) } ).then(
response => {
  if(response.ok) {
    return response.json();
  }
  throw new Error ('Request failed!');
},
networkError => {
  console.log(networkError.message);
}
).then(jsonResponse => jsonResponse);
```

Console object

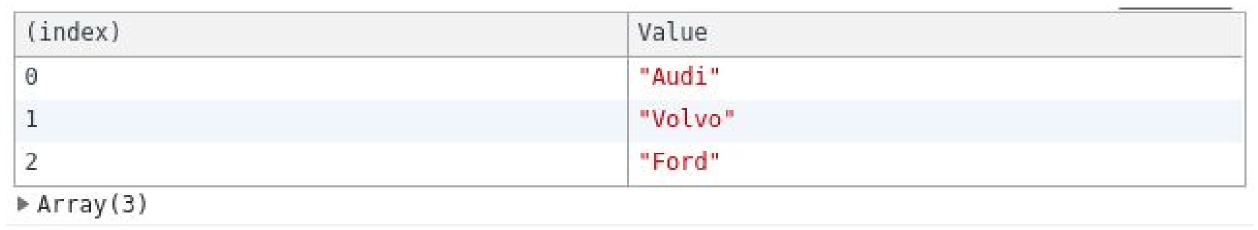
console.clear() method

Clear all messages in the console

console.table() method

Ex 1:

```
console.table(["Audi", "Volvo", "Ford"]);
```

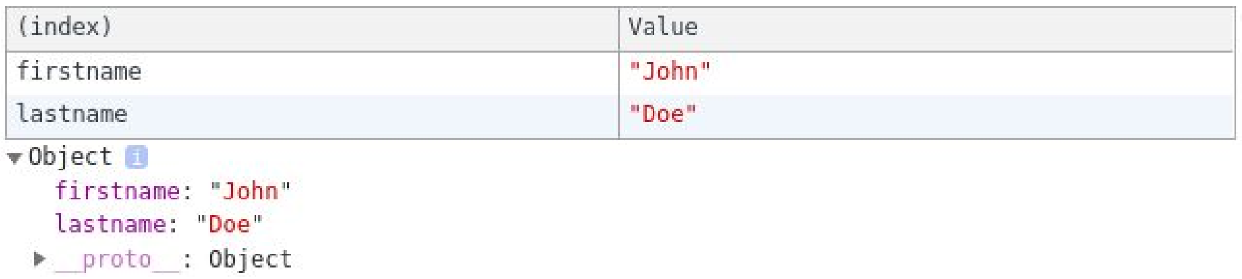


(index)	Value
0	"Audi"
1	"Volvo"
2	"Ford"

```
▶ Array(3)
```

Ex 2:

```
console.table({ firstname : "John", lastname : "Doe" });
```

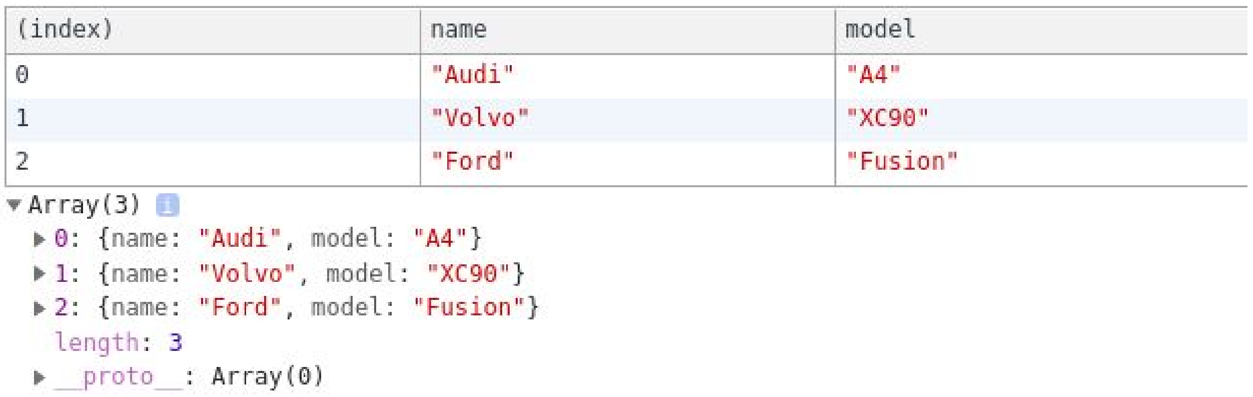


(index)	Value
firstname	"John"
lastname	"Doe"

```
▼Object
   firstname: "John"
   lastname: "Doe"
  ▶__proto__: Object
```

Ex:3

```
var car1 = { name : "Audi", model : "A4" }
var car2 = { name : "Volvo", model : "XC90" }
var car3 = { name : "Ford", model : "Fusion" }

console.table([car1, car2, car3]);
```

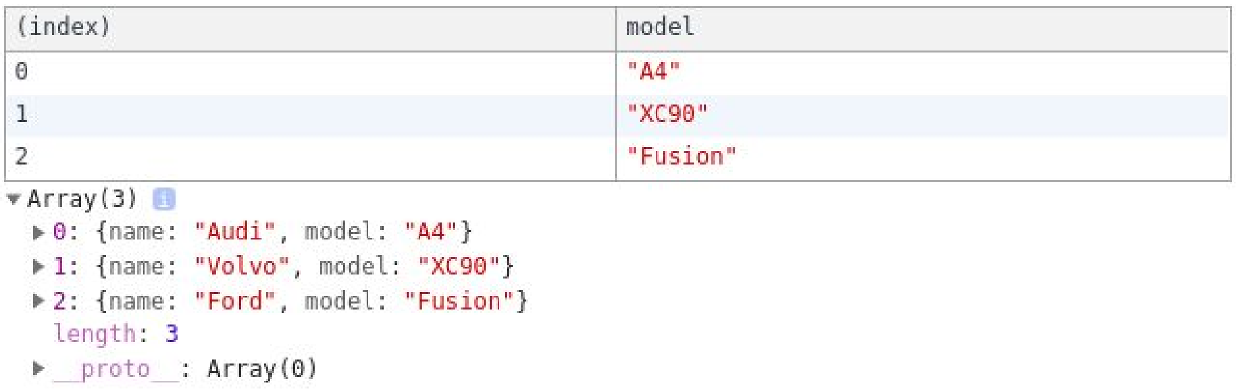


(index)	name	model
0	"Audi"	"A4"
1	"Volvo"	"XC90"
2	"Ford"	"Fusion"

```
▼Array(3)
  ▶0: {name: "Audi", model: "A4"}
  ▶1: {name: "Volvo", model: "XC90"}
  ▶2: {name: "Ford", model: "Fusion"}
   length: 3
  ▶__proto__: Array(0)
```

To Specifying that we only want to include the "model" column in the table:

```
var car1 = { name : "Audi", model : "A4" }
var car2 = { name : "Volvo", model : "XC90" }
var car3 = { name : "Ford", model : "Fusion" }

console.table([car1, car2, car3], ["model"]);
```



(index)	model
0	"A4"
1	"XC90"
2	"Fusion"

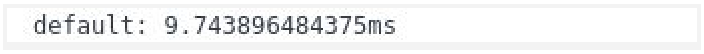
```
▼Array(3)
  ▶0: {name: "Audi", model: "A4"}
  ▶1: {name: "Volvo", model: "XC90"}
  ▶2: {name: "Ford", model: "Fusion"}
   length: 3
  ▶__proto__: Array(0)
```

Console.time() & console.timeEnd()

The console.time() method starts the timer and the console.timeEnd() method ends the timer and writes the result in the console view.

Ex: 1

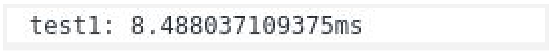
```
console.time();
for (i = 0; i < 100000; i++) {
  // some code
}
console.timeEnd();
```



```
default: 9.743896484375ms
```

Ex: 2

```
var i;
console.time("test1");
for (i = 0; i < 100000; i++) {
  // some code
}
console.timeEnd("test1");
```

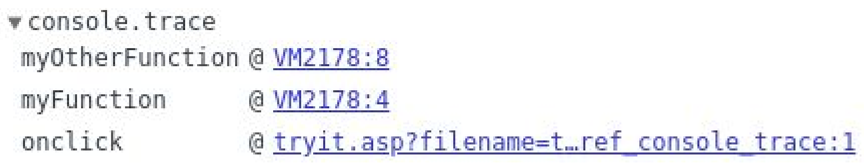


```
test1: 8.488037109375ms
```

console.trace()

Show the trace of how the code ended up here:

```
<button onclick="myFunction()">Start Trace</button>

<script>

function myFunction() {
  myOtherFunction();
}

function myOtherFunction() {
  console.trace();
}

</script>
```



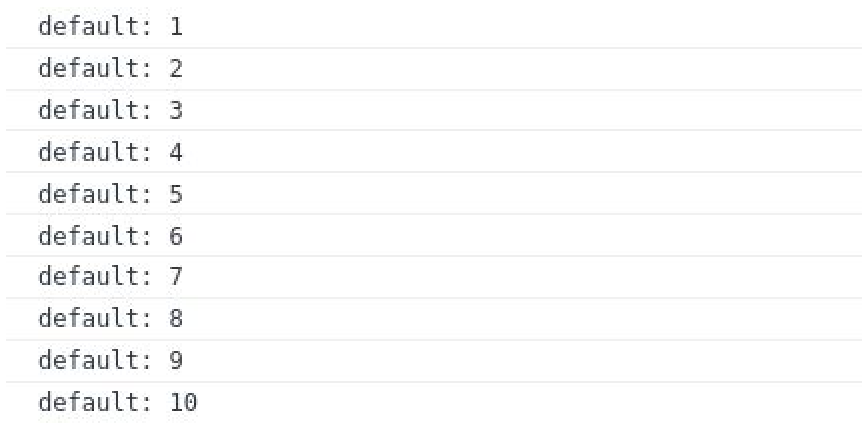
```
▾console.trace
 myOtherFunction @ VM2178:8
 myFunction      @ VM2178:4
 onclick         @ tryit.asp?filename=t…ref_console_trace:1
```

console.count()

Write to the console the number of time the console.count() is called inside the loop:

Ex: 1

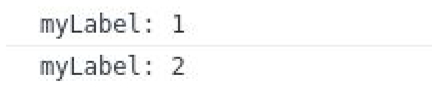
```
<script>

for (i = 0; i < 10; i++) {
  console.count();
}

</script>
```



```
default: 1
default: 2
default: 3
default: 4
default: 5
default: 6
default: 7
default: 8
default: 9
default: 10
```

Ex: 2

```
<script>

console.count("myLabel");
console.count("myLabel");

</script>
```

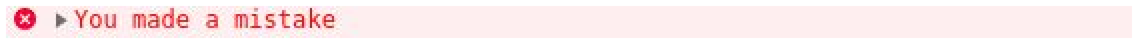


```
myLabel: 1
myLabel: 2
```

console.error ()

Write an error to the console:

```
console.error("You made a mistake");
```

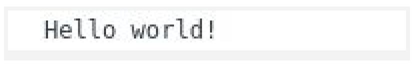


```
▶You made a mistake
```

console.log()

Write message to the console

```
console.log("Hello world!");
```



```
Hello world!
```

-