OBJECT METHOD:

INSIDE DE OBJECT DECLARATION

let user = {

name: "John",

age: 30,

sayHi() {

// "this" is the "current object"

alert(this.name);

}

};

user.sayHi(); // John

A PROPERTY METHOD

let user = { name: "John" };

let admin = { name: "Admin" };

function sayHi() {

alert( this.name );

}

// use the same function in two objects

user.f = sayHi;

admin.f = sayHi;

// these calls have different this

// "this" inside the function is the object "before the dot"

user.f(); // John (this == user)

admin.f(); // Admin (this == admin)

admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)

let user = {

firstName: "Ilya",

sayHi() {

let arrow = () => alert(this.firstName);

arrow();

}

};

user.sayHi(); // Ilya

**[Summary](https://javascript.info/object-methods" \l "summary)**

* Functions that are stored in object properties are called “methods”.
* Methods allow objects to “act” like object.doSomething().
* Methods can reference the object as this.

The value of this is defined at run-time.

* When a function is declared, it may use this, but that this has no value until the function is called.
* A function can be copied between objects.
* When a function is called in the “method” syntax: object.method(), the value of this during the call is object.

Please note that arrow functions are special: they have no this. When this is accessed inside an arrow function, it is taken from outside.

function makeUser() {

return {

name: "John",

ref() {

return this;

}

};

}

let user = makeUser();

alert( user.ref().name ); // John

Now it works, because user.ref() is a method. And the value of this is set to the object before dot ..

* read() prompts for two values and saves them as object properties with names a and b respectively.
* sum() returns the sum of saved values.
* mul() multiplies saved values and returns the result.
* let calculator = {
* sum() {
* return this.a + this.b;
* },
* mul() {
* return this.a \* this.b;
* },
* read() {
* this.a = +prompt('a?', 0);
* this.b = +prompt('b?', 0);
* }
* };
* calculator.read();
* alert( calculator.sum() );
* alert( calculator.mul() );

let ladder = {

step: 0,

up() {

this.step++;

},

down() {

this.step--;

},

showStep: function() { // shows the current step

alert( this.step );

}

};

ladder.up();

ladder.up();

ladder.down();

ladder.showStep(); // 1

ladder.down();

ladder.showStep(); // 0

Chaining

let ladder = {

step: 0,

up() {

this.step++;

return this;

},

down() {

this.step--;

return this;

},

showStep() {

alert( this.step );

return this;

}

};

ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0

ladder

.up()

.up()

.down()

.showStep() // 1

.down()

.showStep(); // 0

QUESTIONS:

* When a function is called in the “method” syntax: object.method(), the value of this during the call is object.

I DON’T UNDERSTAND THAT

* I don’t understand chaining. When is it useful? What does it mean to return this? Why is it different from the other exercise?

OBJECTS:

OBJECT LITERALS:

An object literal is an object that is created directly in the language by wrapping all its properties and methods in curly braces {}.

 objects to be created quickly without the need for defining a class.

A SUPER EXAMPLE:

*const* superman = {

name: 'Superman',

'real name': 'Clark Kent',

height: 75,

weight: 235,

hero: true,

villain: false,

allies: ['Batman','Supergirl','Superboy'],

fly() {

*return* 'Up, up and away!';

}};

Each property is a key-value pair, separated by commas.

 The property 'real name' in the example above needs to be quoted because it contains a space.

All objects are mutable at any time when a program is running. This means that its properties and methods can be changed or removed, and new properties and methods can be added to the object, even if it was declared using const.

CREATING OBJECTS:

To create an object literal, simply enter a pair of curly braces.

*const* spiderman = {}; //LITERAL

*const* spiderman = *new* Object(); // CONSTRUCTOR METHOD

*const* name = 'Iron Man';

*const* realName = 'Tony Stark';

*// long wayconst* ironMan = { name: name, realName: realName };

*// short ES6 wayconst* ironMan = { name, realName };

ES6 provided a shorthand method of creating objects if a property key is the same as a variable name that the property value is assigned to

ACCESSING PROPERTIES:

Dot notation is much more common, but bracket notation has a few advantages: it's the only way to access nonstandard property and method names that don’t follow the variable naming rules. It also lets you evaluate an expression and use it as the property key:

superman["real" + " " + "name"] *// the property is built using string concatenation*

<< "Clark Kent"

COMPUTED PROPERTIES:

 This means that JavaScript code can be placed inside square brackets and the property key will be the return value of that code.

const hulk = { name: 'Hulk', ['catch' + 'Phrase']: 'Hulk Smash!' };

The value of a property has always been allowed to be a JavaScript expression.

The new Symbol date type can also be used as a computed property key:

QUESTION:

I DON’T UNDERSTAND HOW SYMBOLS WORK. WHAT ARE THEY USEFUL FOR?

CALLING METHODS:

superman.fly()<< 'Up, up and away!'

superman['fly']()<< 'Up, up and away!'

CHECKING IF PROPERTIES AND METHODS EXIST:

The in operator can be used to check whether an object has a particular property.

'city' in superman;

<< false

Alternatively, you could also check to see if the property or method doesn’t return undefined:

superman.city !== undefined;

<< false

Another way is to use the hasOwnProperty() method. As mentioned earlier, objects can inherit properties from other objects, so all objects have a method called hasOwnProperty(). This can be used to check whether an object has a property that is its own, rather than one that has been inherited from another object:

superman.hasOwnProperty('city');

<< false

superman.hasOwnProperty('name');

<< true

This method will only return any properties that belong to that particular object, whereas using in or !== undefined will return true, even if the property has been inherited from another object

FINDING ALL THE PROPERTIES OF AN OBJECT:

for in loop

for(const key in superman) {

console.log(key + ": " + superman[key]);

}

<< "name: Superman"

<< "real name: Clark Kent"

<< "height: 75"

<< "weight: 235"

<< "hero: true"

<< "villain: false"

<< "allies: Batman,Supergirl,Superboy"

<< "fly: function (){

console.log(\"Up, up and away!\");

}"

To make sure that only an object’s own properties are returned, a quick check can be implemented beforehand:

for(const key in superman) {

if(superman.hasOwnProperty(key)){

console.log(key + ": " + superman[key]);

}

}

The Object.keys() method will return an array of all the keys of any object that is provided as an argument. We can then iterate over this array to access all the keys of an object:

for(const key of Object.keys(superman)) {

console.log(key);

}

<< name

real name

height

weight

hero

villain

allies

fly

ES2017 also adds some the Object.values() that works in the same way, but returns an array of all the object's value:

Object.entries() is also part of ES2017 and returns an array of key-value pairs. These key-value pairs are returned in arrays, but they can be destructured and accessed individually using the following notation:

for(const [key,value] of Object.entries(superman)) {

console.log(`${key}: ${value}`);

}

<< name: Superman

real name: Clark Kent

height: 75

weight: 235

hero: true

villain: false

allies: [ 'Batman','Supergirl','Superboy' ]

fly: [Function: fly]

ADDING PROPERTIES:

superman.city = 'Metropolis';<< 'Metropolis'

It's important to note that properties don’t always appear in the order they were entered.

CHANGING PROPERTIES:

superman['real name'] = 'Kal-El';<< 'Kal-El'

REMOVING PROPERTIES:

*delete* superman.fly

<< true

NESTED OBJECTS:

const jla = {

superman: { realName: 'Clark Kent' },

batman: { realName: 'Bruce Wayne' },

wonderWoman: { realName: 'Diana Prince" },

flash: { realName: 'Barry Allen' },

aquaman: { realName: 'Arthur Curry' },

}

jla.wonderWoman.realName

<< "Diana Prince"

jla['flash']['realName']

<< "Barry Allen"

jla.aquaman['realName']

<< "Arthur Curry"

An important concept to get your head around is that objects are assigned by reference. This means that if a variable is assigned to an object that already exists, it will simply point to the exact same space in memory. So any changes made using either reference will affect the same object.

OBJECTS AS PARAMETERS TO FUNCTIONS:

An object literal can be passed as a parameter to a function. This is useful when there are a large number of parameters, as it allows the arguments to be provided by name and in any order. This means you don't have to remember the order to enter them when invoking a function.

*function* greet({greeting,name,age}) {

*return* `${greeting}! My name is ${name} and I am ${age} years old.`;}

QUESTION:

THE BRACKETS INSIDE THE FUNCTION ARE CREATING AN OBJECT LITERAL? IS IT TO LET US KNOW THAT THE PARAMETER WOULD BE AN OBJECT? WHAT IF I WANT TO PASS THE NAME OF THE OBJECT AS A PARAMETER INSTEAD OF A ANY PART OF IT?

This technique is referred to as using **named parameters** and is often used when a function has a large amount of optional parameters.

Here’s an example of how the function can be used. Notice how the order of the properties in the argument object differs from the order they are listed in the object provided as a parameter to the function:

greet({ greeting: `What's up dude`, age: 10, name: `Bart` });

<< 'What\'s up dude! My name is Bart and I am 10 years old.'

We can provide default values for some of the parameters using assignment,

*function* greet({greeting='Hello',name,age=18}) {

*return* `${greeting}! My name is ${name} and I am ${age} years old.`;}

THIS:

const dice = {

sides: 6,

roll() {

return Math.floor(this.sides \* Math.random()) + 1;

}

}

NAMESPACING:

This is done by creating an object literal that serves as the namespace, then adding any values as properties of that object, and any functions as methods.

const myMaths = {

square(x) {

return x \* x;

},

mean(array,callback) {

if (callback) {

array.map( callback );

}

const total = array.reduce((a, b) => a + b);

return total/array.length;

}

};

Now these functions need to be preceded by the namespace to be invoked:

myMaths.square(3)

<< 9

myStats.mean([1,2,3])

<< 2

BUILT-IN OBJECTS:

JSON:

1. Property names must be double-quoted
2. Permitted values are double-quoted strings, numbers, true, false, null, arrays and objects
3. Functions are not permitted values

const batman = '{"name": "Batman","real name": "Bruce Wayne","height": 74, "weight": 210, "hero": true, "villain": false, "allies": ["Robin","Batgirl","Superman"]}'

The parse() method takes a string of data in JSON format and returns a JavaScript object:

JSON.parse(batman);

<< { name: 'Batman',

'real name': 'Bruce Wayne',

height: 74,

weight: 210,

hero: true,

villain: false,

allies: [ 'Robin', 'Batgirl', 'Superman' ] }

The stringify() method does the opposite, taking a JavaScript object and returning a string of JSON data, as can be seen in the example:

const wonderWoman = {

name: 'Wonder Woman',

'real name': 'Diana Prince',

height: 72,

weight: 165,

hero: true,

villain: false,

allies: ['Wonder Girl','Donna Troy','Superman'],

lasso: function(){

console.log('You will tell the truth!');

}

}

JSON.stringify(wonderWoman);

<< '{"name":"Wonder Woman","real name":"Diana Prince","height":72,

"weight":165,"hero":true,"villain":false,"allies":["Wonder Girl",

"Donna Troy","Superman"]}'

Note that the lasso method is simply ignored by the stringify() method. Functions are not allowed in json.

You can also add a space argument that will add new lines between each key-value pair, which is useful when displaying the results in a browser:

JSON.stringify(wonderWoman, null, " ");

<< '{\n "name": "Wonder Woman",\n "real name": "Diana Prince",\n "height": 72,\n "weight": 165,\n "hero": true,\n "villain": false,\n "allies": [\n "Wonder Girl",\n "Donna Troy",\n "Superman"\n ]\n}'

THE MATH OBJECT:

MATHEMATICAL CONSTANTS:

The Math object has eight properties that represent a mix of commonly used math constants. Note that they are all named in capital letters, as is the convention for constant values:

Math.PI // The ratio of the circumference and diameter of a circle

<< 3.141592653589793

Math.SQRT2 // The square root of 2

<< 1.4142135623730951

Math.SQRT1\_2 // The reciprocal of the square root of 2

<< 0.7071067811865476

Math.E // Euler's constant

<< 2.718281828459045

Math.LN2 // The natural logarithm of 2

<< 0.6931471805599453

Math.LN10 // The natural logarithm of 10

<< 2.302585092994046

Math.LOG2E // Log base 2 of Euler's constant

<< 1.4426950408889634

Math.LOG10E // Log base 10 of Euler's constant

<< 0.4342944819032518

MATHEMATICAL METHODS:

The Math.abs() method returns the absolute value of a number.

Math.abs(3);

<< 3

Math.abs(-4.6);

<< 4.6

ROUNDING METHODS:

The Math.ceil() method will round a number up to the next integer, or remain the same if it is already an integer:

The Math.floor() method will round a number down to the next integer, or remain the same if it is already an integer:

The Math.round() method will round a number to the nearest integer:

ES6 also introduced the Math.trunc() method that returns the integer-part of a number – that is, it gets truncated at the decimal point:

POWERS AND ROOTS:

The Math.exp() method will raise a number to the power of Euler’s constant:

The Math.pow() method will raise any number (the first argument) to the power of another number (the second argument):

The Math.sqrt() method returns the positive square root of a number:

The Math.cbrt() method was introduced in ES6, which returns the cube root of numbers:

The Math.hypot() method was also introduced in ES6. It returns the square root of the sum of the squares of all its arguments. This can be used to calculate the hypotenuse of a right-angled triangle:

Math.hypot(3,4); // returns the square root of 3 squared + 4 squared

<< 5

Math.hypot(2,3,6); // more than 2 arguments can be used

<< 7

LOGARITHMIC METHODS:

The Math.log() method returns the natural logarithm of a number:

Math.log2(8); *// 8 is 2 to the power of 3*<< 3

Math.log10(1000000); *// 1 million is 10 to the power 6*<< 6

MAXIMUM AND MINIMUM METHODS:

The Math.max() method returns the maximum number from its arguments:

And the Math.min() method unsurprisingly returns the minimum number from the given arguments:

TRIGONOMETRIC FUNCTIONS:

Math.sin()

Math.cos()

Math.tan()

Math.asin()

Math.acos()

Math.atan()Methods for the [hyperbolic functions](https://en.wikipedia.org/wiki/Hyperbolic_function), sinh(), cosh() and tanh() were also added in ES6, as well as their inverses:

RANDOM NUMBERS:

The Math.random() method is used to create random numbers, which can be very useful when writing programs. Calling the method will generate a number between 0 (inclusive) and 1 (exclusive), like so:

To generate a random number between 0 and another number, we can multiply the value by that number. The following code generates a random number between 0 and 6:

6 \* Math.random();<< 4.580981240354013

If we want to generate a random integer, we can use the Math.floor() method that we saw earlier to remove the decimal part of the return value. The following code generates a random integer between 0 and 5 (it will never be 6, because it always rounds down):

Math.floor(6 \* Math.random());<< 4

QUESTION:

WHAT IF I WANT TO GENERATE A RANDOM NUMBER BETWEEN 5 AND 10?

THE DATE OBJECT:

Each object represents a single moment in time.

*const* today = *new* Date(); WE CAN ONLY CREATE IS USING A CONSTRUCTOR FUNCTION

today.toString(); TO SEE THE TODAY DATA

<< 'Tue Feb 14 2017 16:35:18 GMT+0000 (GMT)'

If an argument is not supplied, the date will default to the current date and time. It’s possible to create Date objects for any date by supplying it as an argument to the constructor function. This can be written as a string in a variety of forms:

const christmas = new Date('2017 12 25');

christmas.toString();

<< 'Mon Dec 25 2017 00:00:00 GMT+0000 (GMT)'

const chanukah = new Date('12 December 2017');

// First day of Chanukah

chanukah.toString();

<< 'Tue Dec 12 2017 00:00:00 GMT+0000 (GMT)'

const eid = new Date('Sunday, June 25, 2017');

// Eid-al-Fitr

eid.toString();

<< 'Sun Jun 25 2017 00:00:00 GMT+0100 (BST)'

As you can see, the string passed to the Date constructor can be in a variety of formats. However, in order to be more consistent, it’s better to provide each bit of information about the date as a separate argument. The parameters that can be provided are as follows:

new Date(year,month,day,hour,minutes,seconds,milliseconds)

example:

const solstice = new Date(2017, 5, 21);

// Summer Solstice

solstice.toString();

<< 'Wed Jun 21 2017 00:00:00 GMT+0100 (BST)'

An alternative is to use a timestamp, which is a single integer argument that represents the number of milliseconds since the Epoch (1st January 1970):

const diwali = new Date(1508367600000);

diwali.toString();

<< 'Thu Oct 19 2017 00:00:00 GMT+0100 (BST)'

The properties of date objects are unable to be viewed or changed directly. Instead, they have a number of methods known as **getter** methods, which return information about the date object, such as the month and year.

GETTER METHODS:

The getDay() and getUTCDay() methods are used to find the day of the week that the date object falls on. It returns a number, starting at 0 for Sunday, up to 6 for Saturday:

diwali.getDay(); *// it's on a Thursday*<< 4

The getDate() and getUTCDate() methods return the day of the month for the date object (note that these values start counting from 1, not 0, so they return the actual day of the month):

diwali.getDate(); *// it's on the 19th*<< 19

The getMonth() and getUTCMonth() methods can be used to find the month of the date object. It returns an integer, but remember to count from 0; so January is 0, February is 1, and so on up to December being 11:

The getFullYear() and getUTCFullYear() methods return the year of the date object. There is also a getYear() method, but it isn’t Y2K compliant, so shouldn’t be used:

There are also getHours(), getUTCHours(), getMinutes(), getUTCMinutes(), getSeconds(), getUTCSeconds, getMilliseconds(), and getUTCMilliseconds() methods that will return the hours, minutes, seconds and milliseconds since midnight.

The getTime() method returns a timestamp representing the number of milliseconds since the Epoch:

This can be useful for incrementing dates by a set amount of time. For example, a day can be represented by 1000 \* 60 \* 60 \* 24 milliseconds:

*const* christmasEve = *new* Date(christmas.getTime() - 1000 \* 60 \* 60 \* 24) *// one day before Christmas*christmasEve.toString();<< Fri Dec 26 2014 00:00:00 GMT+0000 (GMT)"

The getTimezoneOffset() method returns the difference, in minutes, between the local time on the computer and UTC. For example, my timezone is currently the same as UTC, so it returns 0:

*new* Date().getTimezoneOffset();<< 0

SETTER METHODS:

Most of the getter methods covered in the previous section have equivalent **setter** methods. These are methods that can be used to change the value of the date held in a Date object. Each of the methods takes an argument representing the value to which you update the date. The methods return the timestamp of the updated date object.

As an example, we can change the value of the date stored in the diwali variable so that it contains the date of Diwali in 2018, which is on Wednesday, November 7, 2018:

diwali.setDate(7);

<< 1507330800000

diwali.setMonth(10); // November is month 10

<< 1510012800000

diwali.setFullYear(2018);

<< 1541548800000

Note that the values returned by these functions is the timestamp representing the number of milliseconds since the Epoch. To see the actual date, we need to use the toString() method:

There are also setHours(), setUTCHours(), setMinutes(), setUTCMinutes(), setSeconds(), setUTCSeconds, setMilliseconds() and setUTCMilliseconds() methods that can be used to edit the time portion of a Date object.

Alternatively, if you know the date as a timestamp, you can use the setTime() method:

diwali.setTime(1447200000000);

<< 1541548800000

THE REGEXP OBJECT:

between forward slashes that we’ve already seen:

const pattern = /[a-zA-Z]+ing$/; //LITETALS

const pattern = new RegExp('[a-zA-Z]+ing'); //CONSTRUCTOR FUNCTION

Using literal regular expressions takes less typing, but there are advantages to using the constructor function as it lets you create regular expressions using strings, which can be useful when the regular expression is provided from user input; in a form, for example. Constructors also have the advantage of letting you create a regular expression using a variable:

const language = 'JavaScript';

const pattern = new RegExp(language);

REGEXP METHODS:

Once you’ve created a regular expression object, you can use the test() method to see if a string (passed to the method as a parameter) matches the regular expression pattern. It returns true if the pattern is in the string, and false if it isn’t.

We can see an example of the test() method used below, using the same pattern we created earlier that tests if a word ends in 'ing':

pattern.test('joke');

<< false

pattern.test('joking');

<< true

pattern.test('jokingly');

<< false

The exec() method works in the same way as the test() method, but instead of returning true or false, it returns an array containing the first match found, or null if there aren’t any matches:

pattern.exec('joke');

<< null

pattern.exec('joking');

BASIC REGULAR EXPRESSIONS:

At the most basic level, a regular expression will just be a string of characters, so the following will match the string 'JavaScript':

const pattern = /JavaScript/;

<< /JavaScript/

CHARACTER GROUP:

Groups of characters can be placed together inside square brackets. This character group represents any one of the characters inside the brackets. For example, the following regular expression matches any vowel:

const vowels = /[aeiou]/

<< /[aeiou]/

A sequence of characters can also be represented by placing a dash [ - ] between the first and last characters; for example, all the uppercase letters can be represented as:

/[A-Z]/

The digits 0-9 can be represented as

/[0-9]/

If a \^ character is placed at the start of the sequence of characters with the brackets, it negates the sequence, so the following regular expression represents any character that is not a capital letter:

/[^A-Z]/

These groups can be combined with letters to make a more complex pattern. For example, the following regular expression represents the letter J (lowercase or capital) followed by a vowel, followed by a lowercase v, followed by a vowel:

pattern = /[Jj][aeiou]v[aeiou]/;

<< /[Jj][aeiou]v[aeiou]/

pattern.test('JavaScript');

<< true

pattern.test('jive');

<< true

pattern.test('hello');

<< false

REGULAR EXPRESSION PROPERTIES:

Regular expressions are objects, and have the following properties:

* The global property makes the pattern return all matches. By default, the pattern only looks for the first occurrence of a match.
* The ignoreCase property makes the pattern case-insensitive. By default, they are case sensitive.
* The multiline property makes the pattern multiline. By default, a pattern will stop at the end of a line.

The following flags can be placed after a regular expression literal to change the default properties:

* g sets the global property to true
* i sets the ignoreCase property to true
* m sets the multiline property to true

For example, the following regular expression will match 'JavaScript' or 'javascript'

pattern = /java/i<< /java/i

These properties can be checked using the dot notation, but cannot be updated once the regular expression has been created, as can be seen in the following example:

pattern = /java/i

<< /java/i

pattern.ignoreCase // checking it is true

<< true

pattern.ignoreCase = false // this won't work

<< false

pattern.ignoreCase // has it changed? Nope!

<< true

The only way to change the ignoreCase property to false is to redefine the regular expression:

SPECIAL CHARACTER:

In a regular expression, there are a number of characters that have a special meaning, commonly known as metacharacters:

* . matches any character, except line breaks
* \w matches any word character, and is equivalent to [A-Za-z0-9\_]
* \W matches any non-word character, and is equivalent to [\^A-Za-z0-9\_]
* \d matches any digit character, and is equivalent to [0-9]
* \D matches any non-digit character, and is equivalent to [^0-9]
* \s matches any whitespace character, and is equivalent to [ \t\r\n\f]
* \S matches any non-whitespace character, and is equivalent to [^ \t\r\n\f]

MODIFIERS:

Modifiers can be placed after a token to deal with multiple occurrences of that token:

* ? makes the preceding token in the regular expression optional
* \* matches one or more occurrences of the preceding token
* + matches one or more occurrences of the preceding token
* {n} matches *n* occurrences of the preceding token
* {n,} matches at least *n* occurrences of the pattern
* {,m} matches at most *m* occurrences of the preceding token
* {n,m} matches at least *n* and at most *m* occurrences of the preceding token
* ^ marks the position immediately before the first character in the string
* $ marks the position immediately after the last character in the string

Any special characters or modifiers can be escaped using a backslash. So if you wanted to match a question mark, ?, you would need to use the regular expression /\?/.

GREEDY AND LAZY MODIFIERS:

All the modifiers above are greedy, which means they will match the longest possible string. They can be made into lazy modifiers that match the shortest possible string by adding an extra '?' after the modifier.

For example, consider the string 'abracadabra':

const word = 'abracadabra';

The greedy pattern /a.+a/ will return the whole string because it is the longest string that matches the pattern of 'a', followed by numerous characters and finishing with an 'a':

const greedyPattern = /a.+a/;

greedyPattern.exec(word);

<< [ 'abracadabra', index: 0, input: 'abracadabra' ]

A PRACTICAL EXAMPLE:

STRING METHODS:

The following example uses a regular expression to split a string every time there are one or more occurrences of a whitespace character:

'Hello World!'.split(/\s+/) //

<< ['Hello', 'World!']

The match() method returns an array of all the matches. By default, only the first is returned:

'JavaScript'.match(/[aeiou]/); *// return the first vowel*<< ['a']

We can use the g flag to return all the matches:

'JavaScript'.match(/[aeiou]/g); // return an array of all the vowels

<< ['a', 'a', 'i']

The search() method returns the position of the first match:

"I'm learning JavaScript".search(/java/i);<< 13

It returns -1 if there is no match:

The replace() method replaces any matches with another string. The following example will replace all vowels with a '\*' character:

'JavaScript'.replace(/[aeiou]/ig,'\*');<< 'J\*v\*Scr\*pt'

MATCHED GROUPS:

Sub-patterns can be created inside a regular expression by placing them inside parentheses. These are known as *capturing groups*. Any matches to these will then be stored in an array of matches.

Each capturing group is numbered according to the position it appears in the pattern. For example, the first capturing group will be numbered 1, and the second 2, etc. The matches will also be stored in special predefined variables $1, $2 etc.

To demonstrate this, here's an example that searches a string and replaces any HTML anchor tags with [Markdown](https://daringfireball.net/projects/markdown/) notation:

*const* link = "<a href='https://www.sitepoint.com' title='Oh Yeah!'>Awesome Web Resources</a>"

*const* mdLink = link.replace(/<a href='(.\*?)'.\*?>(.\*?)<\/a>/g, "[$2]($1)");

mdLink<< [Awesome Web Resources](https://www.sitepoint.com)

*const* link = "<a href='https://www.sitepoint.com' title='Oh Yeah!'>Awesome Web Resources</a>"

*const* mdLink = link.replace(/<a href='(.\*?)'.\*?>(.\*?)<\/a>/g, "[$2]($1)");

mdLink<< [Awesome Web Resources](<https://www.sitepoint.com>)

**THE DOCUMENT OBJECT MODEL:**

The HTML tag is the root node, and every other part of the document is a child node of this.

We can use JavaScript to access and modify different parts of a web page using a special built-in object called document.

There appear to be some extra #text nodes in this diagram, even in places where there isn't any text. This is because the DOM also stores any whitespace that is in the HTML document as text nodes.

GETTING ELEMENTS:

For example, we can use the nodeType property to find out what type of node it is:

body.nodeType;

<< 1

All nodes have a numerical code to signify what type they are. These are summmarized in the table below.

1 = element

2 = attribute

3 = text

8 = comment

9 = body

We can also use the nodeName property to find the name of the element:

body.nodeName;<< BODY"

LEGACY DOM SHORTCUT METHODS:

There are some methods from DOM Level 0 that can still be employed to access commonly used elements. These include:

* Document.body returns the body element of a web page, as we saw in the previous example.
* Document.images returns a node list of all the images contained in the document.
* Document.links returns a node list of all the <a> elements and <area> elements that have an href attribute.
* Document.anchors returns a node list of all the <a> elements that have a name attribute.
* Document.forms returns a node list of all the forms in the document. This will be used when we cover forms in Chapter 8.
* Node lists are array-like objects, but they are not arrays. You can access each item using index notation. For example, document.images[0] will return the first image in the node list of all the images in the document.
* They also have a length property, which can be used to iterate through every element using a for loop, like so:

ES6 makes it very easy to turn a node list into an array, however. You can either use the Array.from() method:

*const* imageArray = Array.*from*(document.images);

Or you can use the spread operator:

*const* imageArray = [...document.images];

GETTING AN ELEMENT BY ITS ID:

If no element exists with the ID provided, null is returned.

GETTING ELEMENTS BY THEIR TAG NAME:

getElementsByTagName() will return a live node list of all the elements with the tag name that is provided as an argument. For example, we can get all the list items (HTML tag of <li>) in the document using this code:

*const* listItems = document.getElementsByTagName('li');

listItems[0];<< <li *class*='hero'>Superman</li>

listItems[1];<< <li *class*='vigilante hero' id='bats'>Batman</li>

listItems[2];<< <li *class*='hero'>Wonder Woman</li>

GET ELEMENTS BY THEIR CLASS NAME:

getElementsByClassName() will return a live node list of all elements that have the class name that is supplied as an argument. For example, we can return a collection of all elements with the class of 'hero' using the following code:

Note that if there are no elements with the given class, an HTML collection is still returned, but it will have a length of 0:

QUERY SELECTOR:

The document.querySelector() method allows you to use CSS notation to find the first element in the document that matches that matches a CSS selector provided as an argument. If no elements match, it will return null.

The document.querySelectorAll() method also uses CSS notation but returns a node list of all the elements in the document that match the CSS query selector. If no elements match, it will return an empty node list.

CSS query selectors are a powerful way of specifying very precise items on a page. For example, CSS pseudo-selectors can also be used to pinpoint a particular element. The following code, for example, will return only the last list item in the document:

*const* wonderWoman = document.querySelector('li:last-child');

The querySelector() method can be called on any element, rather than just document. For example, we can get a reference to the <ul> element, using the following code:

const ul = document.querySelector('ul#roster');

const batman = ul.querySelector('li#bats')

NAVIGATING THE DOM TREE:

The childNodes property is a list of all the nodes that are children of the node concerned. The following example will return all the child nodes of the element with an id attribute of roster:

const heroes = document.getElementById('roster');

heroes.childNodes

<< NodeList [#text "

", <li class="hero">, #text "

", <li id="bats">, #text "

", <li class="hero">, #text "

", <li class="hero">, #text "

The children property only returns any element nodes that are children of that node, so will ignore any text nodes. Note that this is only supported in Internet Explorer from version 9 onwards:

heroes.children *// this will only contain list items*<< HTMLCollection [<li *class*="hero">, <li id="bats">, <li *class*="hero">, <li *class*="hero">] (4)

heroes.children.length<< 3

The firstChild property returns the first child of a node:

And the lastChild property returns the last child of a node:

Be careful when using these properties ― the first or last child node can often be a text node, even if it’s just an empty string generated by some whitespace (this can be seen in both of the examples above).

The parentNode property returns the parent node of an element. The following code returns the roster node because it’s the parent of the wonderWoman node:

The nextSibling property returns the next adjacent node of the same parent. It will return null if the node is the last child node of that parent:

The previousSibling property returns the previous adjacent node. It will return null if the node is the first child of that parent:

Once again, these methods find the next and previous node, not element, so they will often return a blank text node, as in the examples above.

FINDING THE VALUE OF A NODE:

It clearly contains the text 'Wonder Woman', but this is held in a text node, which is the first child of the <li> element node:

*const* textNode = wonderWoman.firstChild;<< "Wonder Woman"

Now we have a reference to the text node, we can find the text contained inside it using the nodeValue method:

textNode.nodeValue;<< "Wonder Woman"

We can also find this value using the textContent property. This will return the text content of an element as a string:

wonderWoman.textContent<< "Wonder Woman"

Note that Internet Explorer version 8 does not support the textContent property, but has the innerText property, which works in a similar way.

GETTING AND SETTING ATTRIBUTES:

All HTML elements have a large number of possible attributes such as 'class', 'id', src, and 'href'. The DOM has numerous getter and setter methods that can be used to view, add, remove or modify the value of any of these attributes.

wonderWoman.getAttribute('class');<< "hero"

wonderWoman.getAttribute('src');<< *null*

The setAttribute can change the value of an element’s attributes. It takes two arguments: the attribute that you wish to change, and the new value of that attribute.

If an element does not have an attribute, the setAttribute method can be used to add it to the element. For example, we can add an id of 'amazon' to the wonderWoman element:

CLASSES OF AN ELEMENT:

#### **The className Property**

As we’ve seen, we can modify the class name of an element using the setAttribute() method. There is also a className property that allows the class of an element to be set directly. In addition, it can be used to find out the value of the class attribute:

wonderWoman.className;<< "villain"

wonderWoman.className = 'hero'<< "hero"

Changing the className property of an element by assignment will overwrite all other classes that have already been set on the element.

This problem can be avoided by using the classList property instead.

THE CLASSLIST PRPERTY:

The classList property is a list of all the classes an element has.

The add method can be used to add a class to an element without overwriting any classes that already exist. For example, we could add a class of 'warrior' to the wonderWoman element:

The remove method will remove a specific class from an element. For example, we could remove the class of 'warrior' with the following code:

The toggle method is a particularly useful method that will add a class if an element doesn’t have it already, and remove the class if it does have it. It returns true if the class was added and false if it was removed. For example:

The contains method will check to see if an element has a particular class:

wonderWoman.classList.toggle('hero'); *// will remove the 'hero' class*<< false

for internet explorer below 10 version:

function addClass(element,newClass){

if (element.className) {

element.className = element.className + ' ' + newClass;

} else {

element.className = newClass;

}

return element.className;

}

addClass(wonderWoman,'warrior');

<< "hero warrior"

addClass(wonderWoman,'warrior');

<< "hero warrior"

CREATING DYNAMIC MARKUP:

CREATING AN ELEMENT:

The document object has a createElement() method that takes a tag name as a parameter and returns that element. For example, we could create a new list item as a DOM fragment in memory by writing the following in the console:

*const* flash = document.createElement('li');

CREATING A TEXT NODE:

A text node can be created using the document.createTextNode() method. It takes a parameter, which is a string containing the text that goes in the node. Let's create the text to go in our new element:

const flashText = document.createTextNode('Flash');

APPENDING NOTES:

flash.appendChild(flashText);

This can be made simpler by using the textContent property that every element object has. This will add a text node to an element without the need to append it, so the code above could have been written as the following:

const flash = document.createElement('li');

flash.textContent = 'Flash';

A FUNCTION TO CREATE ELEMENTS:

*function* createElement (tag,text) {

*const* el = document.createElement(tag);

el.textContent = text;

*return* el}

ADDING ELEMENTS TO THE PAGE:

The appendChild method is useful as you’ll often want to add a new element to the bottom of a list. But what if you want to place a new element in between two existing elements?

The insertBefore() method will place a new element before another element in the markup.

heroes.insertBefore(aquaman,wonderWoman);

If an element is required to appear in several different places in the document, it would need to be cloned before each insertion.

REMOVE ELEMENTS FROM A PAGE:

An element can be removed from a page using the removeChild() method. This method is called on the parent node and has a single parameter, which is the node to be removed. It returns a reference to the removed node

heroes.removeChild(aquaman);

<< <li>Aquaman</li>

Because we have a reference to the element, we can easily put it back into the document if we need to:

REPLACING ELEMENTS ON A PAGE:

The replaceChild() method can be used to replace one node with another. It’s called on the parent node and has two parameters: the new node and the node that is to be replaced. For example, if we wanted to change the content of the <h1> tag that makes the title of the page, we could replace the text node with a new one, like so:

const h1 = document.getElementById('title');

const oldText = h1.firstChild;

const newText = document.createTextNode('Justice League of America');

h1.replaceChild(newText,oldText);

The innerHTML element property was standardized as part of the HTML5, although it was already supported by all the major browsers. It returns all the child elements of an element as a string of HTML

heroes.innerHTML<<"

<li *class*=\"hero\">Superman</li>

<li *class*=\"vigilante hero\" id=\"bats\">Batman</li>

<li *class*=\"hero\">Wonder Woman</li>

"

h1.innerHTML = 'Suicide Squad';

heroes.innerHTML = '<li>Harley Quinn</li><li>Deadshot</li><li>Killer Croc</li><li>Enchantress</li><li>Captain Boomerang</li><li>Katana</li><li>Slipknot</li>';

This will now remove all the child elements of the <ul> element and replace them with the string of HTML that was provided, as shown in below.

LIVE COLLECTIONS:

The node lists returned by the document.getElementsByClassName() and document.getElementsByTagName() methods are live collections that will update to reflect any changes on the page. For example, if a new element with the class hero is added, or an existing one is removed, the node list updates automatically without having to make another call to the method. Therefore, its use is discouraged for performance reasons, but it can be useful.

UPDATING CSS:

*const* heroes = document.getElementById('roster');*const* superman = heroes.children[0];

superman.style.border = "red 2px solid";<< "red 2px solid"

CAMEL CASE PROPERTIES:

Any CSS property names that are separated by dashes must be written in camelCase notation, so the dash is removed and the next letter is capitalized because dashes are not legal characters in property names.

For example, the CSS property background-color becomes backgroundColor. We can change the color of the superman background to green using this code:

superman.style.backgroundColor = 'blue';

<< "blue"

superman.style['background color'] = 'blue';<< "blue"

DISAPPEARING ACT:

You can hide the superman element with the following code:

superman.style.display = 'none';<< "none"

The element can be made to 'reappear' by changing the display property back to block:

superman.style.display = 'block';<< "block"

CHECKING STYLE PROPERTIES:

The style property can also be used to see what CSS styles have been set on an element, but unfortunately it applies only to inline styles, and styles set using JavaScript. This means it excludes styles from external stylesheets, which is the most common way of setting styles.

There is a function called getComputedStyle() that will retrieve all the style information of an element that is given as a parameter. This is a read-only property, so is only used for finding out information about the style of an element.

As you can see, it returns an object (more specifically, it is a CSSStyleDeclaration object)

USE WITH CAUTION:

A better alternative would be to add a class of 'highlighted':

superman.classList.add('highlighted');

**EVENT LISTENERS:**

document.body.addEventListener("click", doSomething);

Its first parameter is the type of event, and the second is a callback function that is invoked when the event occurs. There is also a third parameter that we'll cover later in the chapter.

*function* doSomething() {alert('You Clicked!');}

addEventListener('click',doSomething);

Note that the parentheses are not placed after the function when it's used as the argument to an event listener; otherwise, the function will actually be called when the event listener is set, instead of when the event happens!

THE EVENT OBJECT:

Whenever an event handler is triggered by an event, the callback function is called. This function is automatically passed an event object as a parameter that contains information about the event.

To see an example of this, change the doSomething() function in the main.js file to this:

*function* doSomething(event){ console.log(event.type);}

TYPES OF EVENT:

THE EVENT TARGET:

The target property returns a reference to the node that fired the event. If you change the doSomething() function to the following, it will show a message in the console telling us the node that was clicked on:

Copy

*function* doSomething(event){ console.log(event.target);}

COORDINATES OF AN EVENT:

The screenX and screenY properties show the number of pixels from the left and top of the screen respectively where the event took place.

The clientX and clientY properties show the number of pixels from the left and top of the client that is being used (usually the browser window).

The pageX and pageY properties show the number of pixels from the left and top, respectively, where the event took place in the *document*. This property takes account of whether the page has been scrolled.

function doSomething(event){

console.log(`screen: (${event.screenX},${event.screenY}), page: (${event.pageX},${event.pageY}), client: (${event.screenX},${event.screenY})`)

}

TYPES OF EVENTS:

MOUSE EVENTS:

We have already seen the click event that occurs when a mouse button is clicked. There are also the mousedown and mouseup events. These both occur before a click event is fired.

There is also the dblclick event, which occurs when the user doubleclicks on the element to which the event listener is attached.

The mouseover event occurs when the mouse pointer is placed over the element to which the event listener is attached, while the mouseout event occurs when the mouse pointer moves away from an element. This example uses both the mouseover and mouseout events to change the color of the third paragraph (with an ID of 'mouse') when the mouse pointer hovers over it, and back again when it moves away from the paragraph:

The mousemove event occurs whenever the mouse moves. It will only occur while the cursor is over the element to which it’s applied.

KEYBOARD EVENTS:

1. The keydown event occurs when a key is pressed and will *continue to occur* if the key is held down.
2. The keypress event occurs after a keydown event but before a keyup event. The keypress event only occurs for keys that produce character input (plus the 'Delete' key). This means that it’s the most reliable way to find out the character that was pressed on the keyboard.
3. The keyup event occurs when a key is released.
4. QUESTION:

WHAT IS THE DIFFERENCE BETWEEN KEYPRESS AND KEYDOWN? THEY SEEM TO BE THE SAME. WHAT IF A USE THEM BOTH AT THE SAME TIME?

To understand the differences in these events, it is important to distinguish between a physical key on the keyboard and a character that appears on the screen. The keydown event is the action of pressing a key, whereas the keypress event is the action of a character being typed on the screen.

Pressing the modifier keys such as Shift, Ctrl, Alt and meta (Cmd on Mac) will fire the keydown and keyup events, but not the keypress event as they don't produce any characters on the screen.

Each of these keyboard events have an key property that returns the printed representation of the key that was pressed, if it has one.

addEventListener('keypress', (event) => console.log(`You pressed the ${event.key} character`));

MODIFIER KEYS:

Pressing the modifier keys such as Shift, Ctrl, Alt and meta (Cmd on Mac) will fire the keydown and keyup events, but not the keypress event as they don't produce any characters on the screen.

The name of the modifier key is still returned by the key property. To see this, edit the event listener we just used to listen for a keydown event instead:

addEventListener('keydown', (event) => console.log(`You pressed the ${event.key} character`));

<< "You pressed the Control character"

All event objects also contains information about whether a modifier key was held down when the key event occurred. The shiftKey, ctrlKey, altKey, and metaKey are all properties of the event object and return true if the relevant key was held down. For example, the following code will check to see if the user pressed the C key while holding down the Ctrl key:

addEventListener('keydown', (event) => {

if (event.key === 'c' && event.ctrlKey) {

console.log('Action canceled!');

}

});

The following code checks to see if the Shift key was held down when the mouse was clicked:

addEventListener('click', (event) => { *if* (event.shiftKey) { console.log('A Shifty Click!'); }});

TOUCH EVENTS:

The touchstart event occurs when a user initially touches the surface.

Be careful when using the touchstart event as it fires as soon as a user touches the screen. They may be touching the screen because they want to zoom in or swipe, and a touchstart event listener could prevent them from doing this.

The click event is often a much safer option as it still fires when the screen is touched, but there’s a slight delay of 300ms, allowing the user time to perform another action with the device. The click event can be thought of as a "tap" in the context of a touch event.

The touchend event occurs when a user stops touching the surface:

The touchmove event occurs after a user has touched the screen then moves around without leaving. It will continue to occur as long as the user is still touching the screen, even if they leave the element to which the event listener is attached.

The touchenter event occurs when a user has already started touching the surface, but then passes over the element to which the event listener is attached.

QUQUESTION:

HOW DOES IT WORK?

The touchleave event occurs when the user is still touching the surface, but leaves the element to which the event listener is attached.

The touchcancel event occurs when a touch event is interrupted, such as a user’s finger moving outside the document window, or too many fingers being used at once. A pop-up dialog will also cancel a touch event.

If you need to implement gestures, it's probably a good idea to use a library such as [Hammer.JS](http://hammerjs.github.io/) or [zingtouch](https://zingchart.github.io/zingtouch/" \t "_blank) that makes events such as swipe, pinch and rotate easy to implement.

TOUCH EVENT PROPERTIES:

Because it’s possible to touch a surface many times at once, touch event objects have a property called touches. This is a list of touch objects that represents all the touches taking place on that device. It has a length property that tells you how many touch points (usually the user's fingers, but could be a stylus) are in contact with the surface. Each touch object in the list can be accessed using index notation. For example, if a user touches the screen with two fingers, events.touches.length would return 2. The first touch object can be accessed using events.touches[0] and the second using events.touches[1].

Each touch object has a number of properties, many similar to the event object, such as touch.screenX and touch.screenY to find the coordinates of the touch point. They have other properties such as touch.radiusX and touch.radiusY, which give an indication of the area covered by the touch, and touch.force, which returns the amount of pressure being applied by the touch as a value between 0 and 1.

Each touch object has a touch.identifier property, a unique ID that can be used to ensure you are dealing with the same touch.

REMOVING EVENT LISTENER:

An event listener can be removed using the removeEventListener() method.

Note that you shouldn't use anonymous functions as an argument to addEventListener() if you want to remove it later. This is because there needs to be a reference to the same function name in the arguments of removeEventListener().

STOPPING DEFAULT BEHAVIOR:

brokenLink.addEventListener('click',(event) => { event.preventDefault(); console.log('Broken Link!');});

Some events do not allow the default behavior to be prevented. This can vary from browser to browser, but each event object has a property called cancellable that returns false if it cannot be prevented.

You can also see if the default behavior has been prevented by checking the defaultPrevented property.

EVENT PROPAGATION:

Bubbling is when the event fires on the element clicked on first, then bubbles up the document tree, firing an event on each parent element until it reaches the root node.

Capturing starts by firing an event on the root element, then propagates downwards, firing an event on each child element until it reaches the target element that was clicked on.

BUBBLING:

CAPTURING:

The addEventListener() method has a third parameter, which is a boolean value that specifies whether capturing should be used or not. It defaults to false, which is why bubbling happens by default. There may be instances when you would rather capture the events instead; for example, you might want events on outer elements to fire before any events fire on the element that was actually clicked on.

To implement capturing instead, change the code to the following:

ulElement.addEventListener('click', (event) =>console.log('Clicked on ul'),true);

liElement.addEventListener('click', (event) =>console.log('Clicked on li'),true);

If you want the event to both capture and bubble, you must set a separate event handler for both cases, like so:

*// capturing*

ulElement.addEventListener('click', (event) =>console.log('Clicked on ul'),true);

liElement.addEventListener('click', (event) =>console.log('Clicked on li'),true);

*// bubbling*

ulElement.addEventListener('click', (event) =>console.log('Clicked on ul'),false );

liElement.addEventListener('click', (event) =>console.log('Clicked on li'),false );

STOPPING THE BUBBLING PHASE:

The bubble phase can be stopped from occurring by adding the event.stopPropagation() method into the callback function. In the following example, the event will fail to propagate as the third argument is false, which stops capturing, and the event.stopPropagation() method is called, which stops bubbling:

liElement.addEventListener('click', (event) => {console.log('clicked on li');event.stopPropagation(); }, false);

Now clicking on the first <li> element will only log one message, since the click event will not propagate to the <ul> element.

EVENT DELEGATION:

If we wanted to attach event listeners to all the <li> tags so they were highlighted when clicked on, it would need more code to add a separate event listener to each element. In this case, there isn't much difference, but imagine if you had a list of 100 elements!

A better way is to attach the event listener to the parent <ul> element, then use the target property to identify the element that was clicked on. Add the following to main.js to see this in action (remember that the highlight() function used the target property):

ulElement.addEventListener('click',highlight);

Now clicking on any list item will highlight that list item as if it was the target of the click event.

This is a useful method if you are adding extra list elements to the DOM dynamically. Any new list elements that are a child of the <ul> element will automatically inherit this event listener, saving you from having to add an event listener every time a new list item is added.